谨以此书献给上海交通大学获得ACM-ICPC世界冠军十周年

ACM国际大学生程序设计竞赛（ACM-ICPC）系列丛书

ACM国际大学生程序设计竞赛算法与实现

俞　勇　主编



清华大学出版社

ACM 国际大学生程序设计竞赛（ACM-ICPC）系列丛书

ACM 国际大学生程序设计竞赛
算法与实现

俞　勇　主　编

清华大学出版社
北　京

内 容 简 介

ACM 国际大学生程序设计竞赛（ACM-ICPC）是国际上公认的水平最高、规模最大、影响最深的计算机专业竞赛，目前全球参与人数达 20 多万。本书作者将 16 年的教练经验与积累撰写成本系列丛书，全面、深入而系统地将 ACM-ICPC 展现给读者。本系列丛书包括《ACM 国际大学生程序设计竞赛：知识与入门》、《ACM 国际大学生程序设计竞赛：算法与实现》、《ACM 国际大学生程序设计竞赛：题目与解读》、《ACM 国际大学生程序设计竞赛：比赛与思考》等 4 册，其中《ACM 国际大学生程序设计竞赛：知识与入门》介绍了 ACM-ICPC 的知识及其分类、进阶与角色、在线评测系统；《ACM 国际大学生程序设计竞赛：算法与实现》介绍了 ACM-ICPC 算法分类、实现及索引；《ACM 国际大学生程序设计竞赛：题目与解读》为各类算法配备经典例题及题库，并提供解题思路；《ACM 国际大学生程序设计竞赛：比赛与思考》介绍了上海交通大学 ACM-ICPC 的训练及比赛，包括训练札记、赛场风云、赛季纵横、冠军之路、峥嵘岁月。

本丛书适用于参加 ACM 国际大学生程序设计竞赛的本科生和研究生，对参加青少年信息学奥林匹克竞赛的中学生也很有指导价值。同时，作为程序设计、数据结构、算法等相关课程的拓展与提升，本丛书也是难得的教学辅助读物。

图书在版编目（CIP）数据

ACM 国际大学生程序设计竞赛：算法与实现/俞勇主编. —北京：清华大学出版社，2013.1
ACM 国际大学生程序设计竞赛（ACM-ICPC）系列丛书
ISBN 978-7-302-29413-9

Ⅰ. ①A…　Ⅱ. ①俞…　Ⅲ. ①程序设计–竞赛–高等学校–教学参考资料　Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字（2012）第 161172 号

责任编辑： 龙启铭
封面设计：
责任校对： 李建庄
责任印制：

出版发行： 清华大学出版社
网　　址：http://www.tup.com.cn, http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮编：100084
社 总 机：010-62770175　　邮购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课 件 下 载：http://www.tup.com.cn,010-

印 刷 者：
装 订 者：
经　　销： 全国新华书店
开　　本： 185mm×230mm　　印　张：17.75　　字　数：445 千字
版　　次： 2013 年 1 月第 1 版　　印　次：2013 年 1 月第 1 次印刷
印　　数： 1～0000
定　　价： 0.00 元

产品编号：047607-01

写在最前面的话

自从上海交通大学2002年第一次、2005年第二次获得ACM国际大学生程序设计竞赛（ACM International Collegiate Programming Contest，简称ACM-ICPC或ICPC）世界冠军以来，总有记者邀请编者撰写冠军之路类的文章，也总有出版社希望编者出版ACM-ICPC竞赛类的书籍，因为没有想清楚怎么写，所以一直没动笔。直到2010年上海交通大学第三次获得ACM-ICPC世界冠军后，编者决定出版一套系列丛书，包括《ACM国际大学生程序设计竞赛：知识与入门》、《ACM国际大学生程序设计竞赛：算法与实现》、《ACM国际大学生程序设计竞赛：题目与解读》及《ACM国际大学生程序设计竞赛：比赛与思考》4册书籍，全面、深入而系统地将ACM-ICPC展现给读者，把上海交通大学十多年来对ACM-ICPC竞赛的感悟分享给读者。

编写此系列丛书的另一个重要原因是ACM-ICPC竞赛在中国大陆的迅猛发展。自从1996年ACM-ICPC引入中国大陆，前六届仅设立1个赛区，目前每年一般设立5个赛区，并已有30所高校承办过亚洲区预赛；参赛学校从不满20所，到如今已达200多所；参赛人数从不到100人，到如今超过12万人次；总决赛名额从起初的3个，到如今已超过15个。同时，中国大陆在ACM-ICPC竞赛上所取得的成绩也举世瞩目。清华大学9次获得总决赛奖牌（3金5银1铜），位居奖牌榜之首，是实力最强、表现最稳定的高校；上海交通大学8次获得总决赛奖牌（4金3银1铜），3次夺得世界冠军，算是目前国内成绩最好的高校；中山大学4次获得总决赛奖牌（2银2铜），在生源不占优势的情况下，这一成绩令人敬佩；复旦大学3次获得总决赛奖牌（1银2铜），是公认的强校；浙江大学2次获得总决赛奖牌（1金1银），1次夺得世界冠军，再次让国人欢欣鼓舞；北京大学1次获得总决赛奖牌（1铜），队员的综合实力堪称一流；最难能可贵的是，华南理工大学也获得过总决赛的奖牌（1铜），它告诉我们，ACM-ICPC不仅仅是“强校”之间的“对话”，只要坚持参与就会斩获成果。另外，至今已有37所大陆高校参加过全球总决赛，且不论成绩如何，他们在赛场上的奋斗亦值得称道。

本系列丛书的第一册《ACM 国际大学生程序设计竞赛：知识与入门》分为三个部分。知识点部分基本涵盖了竞赛中所涉及的主要知识点，包括数学基础、数据结构、图论、计算几何、论题选编、求解策略等六个大类内容。入门与进阶部分介绍了包括如何快速入门、如何提高自身以及团队水平等，主要根据上海交通大学 ACM-ICPC 队多年参赛经验总结而来。在线资源部分对一些常用的在线评测系统和网上比赛进行了介绍。

本系列丛书的第二册《ACM 国际大学生程序设计竞赛：算法与实现》涵盖了大部分 ACM-ICPC 竞赛常用的经典算法，包括数学、图论、数据结构、计算几何、论题选编五个大类，对每个算法的代码实现，都配有接口说明以及简略的算法阐述，并提供算法的完整程序，贴士部分收集了一些实用的知识点及积分表，方便读者查找使用。

本系列丛书的第三册《ACM 国际大学生程序设计竞赛：题目与解读》分为两个部分。例题精讲部分针对第二册《ACM 国际大学生程序设计竞赛：算法与实现》中的算法配备经典例题，并提供细致的解题思路，读者可以通过这一部分学习和掌握算法；海量题库部分按照算法分类罗列出大量习题，并提供相应的题解，读者可以利用这一部分的题目进行训练，更加熟练地运用各类算法。

本系列丛书的第四册《ACM 国际大学生程序设计竞赛：比赛与思考》从 120 多名队员、2400 余篇文档中精心挑选、编纂而成的文集，包括训练札记、赛场风云、赛季纵横、冠军之路、峥嵘岁月，集中展现了上海交通大学 ACM-ICPC 队 16 年的奋斗历程，记载了这些队员为了实现自己的梦想而不懈努力、勇于拼搏的故事。

这是一套全面、系统地学习 ACM-ICPC 竞赛的知识类书籍；

这是一套详尽、深入地熟悉 ACM-ICPC 竞赛的算法及题目的手册类书籍；

这是一套程序设计、数据结构、算法等相关课程的拓展与提升类书籍；

这是一部上海交通大学 ACM-ICPC 队的成长史；

这是一部激励更多学子勇敢追寻并实现自己最初梦想的励志书。

历时 2 年零 5 个月，终于完成了本系列丛书，编者与队员有一种如释重负的感觉，因为我们把出版这套丛书看得很重，这是我们 16 年的经验与积累，希望对广大读者有用。

值此 ACM-ICPC 进入中国大陆 16 周年、上海交通大学获得 ACM-ICPC 世界冠军 10 周年之际，谨以此系列丛书——

纪念我们曾经走过的路、度过的岁月；

献给所有支持、帮助过我们的人……

俞　勇

2012 年 10 月于上海

前　言

在 ACM 国际大学生程序设计竞赛（ACM International Collegiate Programming Contest，ACM-ICPC 或 ICPC）中，实现算法的能力是非常重要的。尤其是对新手来说，在了解到一个新的算法后，有时会对如何实现该算法产生困惑，也许并不能一下想到很好的实现，这时就需要参考一些已有的实现。

另外，ACM-ICPC 比赛中允许选手将一定量的（一般为 25 页）纸质资料带入比赛现场进行参考。队伍往往会将一些相对较难实现的常用算法的代码整理为 SCL（Standard Code Library，标准代码库）带入赛场，在需要的时候可以直接抄写已有代码，既节省时间，也保证了正确性。

因此我们出版这本收集了大量经典常用算法的用 C++语言实现的代码库，希望可以帮助读者学习算法以及准备比赛用的 SCL。

本书分为两个部分。第一部分为代码库，涵盖了大部分比赛常用的经典算法，包括数学、图论、数据结构、计算几何、论题选编五个大类，对每个算法的代码实现，都配有接口说明以及简略的算法阐述，便于读者理解。第二部分为贴士，收集了一些实用的知识点以及积分表，适合于带入赛场进行参考。

本书编写工作历时两年左右，参与编写工作的人员全部为上海交通大学 ACM-ICPC 队的现役队员。代码大多来自于往年上海交通大学 ACM-ICPC 队使用的 SCL 以及队员的日常训练。同时，本书的编写也得到上海交通大学 ACM-ICPC 队的退役队员大力帮助，他们参与了代码库的收集、整理、校验等工作。

参与本书写稿、审稿的人员主要有（按姓氏笔画为序）：尹天蛟，乌辰洋，任春旭，刘奇，刘彦，寿鹤鸣，李说，杨思逸，吴卓杰，张捷钧，陈明骋，陈泽佳，陈彬毅，陈爽，林承宇，金斌，郑曌，胡张广达，郭晓旭，曹雪智，康南茜，章雍哲，商静波，彭上夫，谭天，缪沛晗，瞿钧等。

在此，衷心感谢所有为此书出版做出直接或间接贡献的人！也真心祝愿此书能够在算法实现和 SCL 的准备上给读者带来帮助。

由于时间仓促，作者水平有限，疏漏、不当和不足之处在所难免，真诚地希望专家和读者朋友们不吝赐教。如果您能在阅读和使用此书过程中发现任何问题或有任何建议，恳请发邮件至 yyu@cs.stu.edu.cn，我们将不胜感激。

编　者

2012 年 10 月于上海

目　录

第一部分　算　法

第一部分

算　法

第 1 章　数　学

1.1 矩　阵

1.1.1 矩阵类

【任务】

实现矩阵的基本变换。

【接口】

结构体：*Matrix*

成员变量：

int n, m	矩阵大小
int $a[\,][\,]$	矩阵内容

重载运算符：+、 − 、 ×

成员函数：

void clear()	清空矩阵

【代码】

```
const int MAXN=1010;
const int MAXM=1010;
struct Matrix{
    int n,m;
    int a[MAXN][MAXM];
    void clear(){
        n=m=0;
        memset(a,0,sizeof(a));
    }
    Matrix operator +(const Matrix &b) const{
        Matrix tmp;
        tmp.n=n; tmp.m=m;
```

```
        for (int i=0; i<n; ++i)
            for (int j=0; j<m; ++j)
                tmp.a[i][j]=a[i][j]+b.a[i][j];
        return tmp;
    }
    Matrix operator -(const Matrix &b) const{
        Matrix tmp;
        tmp.n=n; tmp.m=m;
        for (int i=0; i<n; ++i)
            for (int j=0; j<m; ++j)
                tmp.a[i][j]=a[i][j]-b.a[i][j];
        return tmp;
    }
    Matrix operator *(const Matrix &b) const{
        Matrix tmp;
        tmp.clear();
        tmp.n=n; tmp.m=b.m;
        for (int i=0; i<n; ++i)
            for (int j=0; j<b.m; ++j)
                for (int k=0; k<m; ++k)
                    tmp.a[i][j]+=a[i][k]*b.a[k][j];
        return tmp;
    }
};
```

【使用范例】

参见程序 POJ3420.CPP。

1.1.2 Gauss 消元

【任务】

给一个n元一次方程组，求它们的解集。

【说明】

将方程组做成矩阵形式，再利用三种初等矩阵变换，得到上三角矩阵，最后回代得到解集。

【接口】

int solve(double a[ ][MAXN], bool l[ ], double ans[ ], const int& n);

复杂度：$O(n^3)$

输　入：a　　方程组对应的矩阵

　　　　n　　未知数个数

　　　　l, ans　　存储解，$l[\,]$表示是否为自由元

输　出：解空间的维数

【代码】

```
inline int solve(double a[][MAXN], bool l[], double ans[],
const int& n) {
    int res = 0, r = 0;
    for (int i = 0; i < n; ++i)
        l[i] = false;
    for (int i = 0; i < n; ++i) {
        for (int j = r; j < n; ++j)
            if (fabs(a[j][i]) > EPS) {
                for (int k = i; k <= n; ++k)
                    swap(a[j][k], a[r][k]);
                break;
            }
        if (fabs(a[r][i]) < EPS) {
            ++res;
            continue;
        }
        for (int j = 0; j < n; ++j)
            if (j != r && fabs(a[j][i]) > EPS) {
                double tmp = a[j][i] / a[r][i];
                for (int k = i; k <= n; ++k)
                    a[j][k] -= tmp * a[r][k];
            }
        l[i] = true, ++r;
    }
    for (int i = 0; i < n; ++i)
        if (l[i])
            for (int j = 0; j < n; ++j)
                if (fabs(a[j][i]) > 0)
                    ans[i] = a[j][n] / a[j][i];
    return res;
}
```

【使用范例】

参见程序 POJ1830.CPP。

1.1.3 矩阵的逆

【任务】

给一个矩阵，求它的逆。

【说明】

将原矩阵A和一个单位矩阵E作成大矩阵(A, E)，用初等行变换将大矩阵中的A变为E，则会得到(E, A^{-1})的形式。

【接口】

void inverse(vector<double> A[ ], vector<double> C[ ], int N);

复杂度：$O(n^3)$

输　入：A　原矩阵

C　逆矩阵

N　矩阵的阶数

【代码】

```
inline vector<double> operator * (vector<double> a, double b) {
    int N = a.size();
    vector<double> res(N, 0);
    for (int i = 0; i < N; ++i)
        res[i] = a[i] * b;
    return res;
}
inline vector<double> operator - (vector<double> a, vector<double> b){
    int N = a.size();
    vector<double> res(N, 0);
    for (int i = 0; i < N; ++i)
        res[i] = a[i] - b[i];
    return res;
}
inline void inverse(vector<double> A[], vector<double> C[], int N) {
    for (int i = 0; i < N; ++i)
        C[i] = vector<double>(N, 0);
```

```
    for (int i = 0; i < N; ++i)
        C[i][i] = 1;
    for (int i = 0; i < N; ++i) {
        for (int j = i; j < N; ++j)
            if (fabs(A[j][i]) > 0) {
                swap(A[i], A[j]);
                swap(C[i], C[j]);
                break;
            }
        C[i] = C[i] * (1 / A[i][i]);
        A[i] = A[i] * (1 / A[i][i]);
        for (int j = 0; j < N; ++j)
            if (j != i && fabs(A[j][i] > 0)) {
                C[j] = C[j] - C[i] * A[j][i];
                A[j] = A[j] - A[i] * A[j][i];
            }
    }
```

【使用范例】

参见程序 POJ1166.CPP。

1.1.4　常系数线性齐次递推

【任务】

已知$f_x = a_0 f_{x-1} + a_1 f_{x-2} + \cdots + a_{n-1} f_{x-n}$和$f_0, f_1, \cdots, f_{n-1}$，给定$t$，求$f_t$。

【说明】

f的递推可以看成一个$n \times n$的矩阵$\boldsymbol{A}$乘以一个n维列向量$\boldsymbol{\beta}$，因为矩阵乘法满足结合律，用快速幂可以加速。其中，

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & & 0 \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ a_{n-1} & a_{n-2} & a_{n-3} & & a_0 \end{bmatrix}, \quad \boldsymbol{\beta} = \begin{bmatrix} f_{x-n} \\ f_{x-n+1} \\ \vdots \\ f_{x-2} \\ f_{x-1} \end{bmatrix}$$

【接口】

int solve(int a[ ], int b[ ], int n, int t);

复杂度：$O(n^3 \log t)$

输　入：a　常系数数组
　　　　b　初值数组
　　　　n　数组大小
　　　　t　要求解的项数
输　出：函数在第t项的值f_t
调用外部函数：
　　矩阵类：参见 1.1.1 节

【代码】

```
int solve(int a[], int b[], int n, int t) {
    Matrix M, F, E;
    M.clear(), F.clear(), E.clear();
    M.n = M.m = n;
    E.n = E.m = n;
    F.n = n, F.m = 1;
    for (int i = 0; i < n - 1; ++i)
        M.a[i][i + 1] = 1;
    for (int i = 0; i < n; ++i) {
        M.a[n - 1][i] = a[i];
        F.a[i][0] = b[i];
        E.a[i][i] = 1;
    }
    if (t < n)
        return F.a[t][0];
    for (t -= n - 1; t; t /= 2) {
        if (t & 1)
            E = M * E;
        M = M * M;
    }
    F = E * F;
    return F.a[n - 1][0];
}
```

【使用范例】

参见程序 POJ3070.CPP。

1.2 整除与剩余

1.2.1 欧几里得算法

【任务】

求两个数a,b的最大公约数$\gcd(a,b)$。

【说明】

由贝祖定理得，$\gcd(a,b)=\gcd(b,a-b)$，其中$a \geqslant b$。通过这样不断的迭代，直到$b=0$，a就是原来数对的最大公约数。考虑到只使用减法会超时，我们观察到如果$a-b$仍然大于b的话，要进行一次同样的操作，就把a减到不足b为止，所以有$\gcd(a,b)=\gcd(b,a \bmod b)$。由此可以在log的时间内求出两个数的gcd。

【接口】

int gcd(int a, int b);

复杂度：O(log$N$)，其中 N 和 a,b 同阶

输　入：a,b　两个整数

输　出：a,b的最大公约数

【代码】

```
int gcd(int a, int b) {
    return b == 0? a : gcd(b, a % b);
}
```

【使用范例】

参见程序 GCD.CPP。

1.2.2 扩展欧几里得

【任务】

求出A,B的最大公约数，且求出X，Y满足$AX+BY=\mathrm{GCD}(A,B)$。

【说明】

要求X,Y，满足：$AX+BY=\mathrm{GCD}(A,B)$。

当$B=0$时，有$X=1,Y=0$时等式成立。

当$B>0$时，在欧几里得算法的基础上，已知：

$$\mathrm{GCD}(A,B)=\mathrm{GCD}(B,A \bmod B)$$

先递归求出X',Y'满足：

$$BX' + (A \bmod B)Y' = \mathrm{GCD}(B,A \bmod B) = \mathrm{GCD}(A,B)$$

然后可以回推，我们将上式化简得：

$$BX' + (A - A/B \times B)Y' = \mathrm{GCD}(A,B)$$

$$AY' + BX' - (A/B) \times BY' = \mathrm{GCD}(A,B)$$

这里除法指整除。把含B的因式提取一个B，可得：

$$AY' + B(X' - A/B \times Y') = \mathrm{GCD}(A,B)$$

故$X = Y', Y = X' - A/B \times Y'$。

【接口】

int extend_gcd(int a, int b, int &x, int & y);

复杂度：O(log$N$)，　其中N和a,b同阶。

输　入：a,b　两个整数

　　　　$\&x,\&y$　引用，$ax + by = \mathrm{GCD}(a,b)$的一组解

输　出：a和b的最大公约数

调用后x,y满足方程$ax + by = \mathrm{GCD}(a,b)$。

【代码】

```
int extend_gcd( int a, int b, int &x, int &y ){
    if ( b==0 ){
        x=1;y=0;
        return a;
    } else {
        int r=extend_gcd(b,a%b,y,x);
        y-=x*(a/b);
        return r;
    }
}
```

【使用范例】

参见程序 POJ1006.CPP，POJ2115.CPP。

1.2.3 单变元模线性方程

【任务】

已知a,b,n，求x，使得$ax \equiv b \pmod{n}$。

【说明】

令$d = \gcd(a,n)$，先使用扩展欧几里德求$ax + ny = d$的解。如果b不能整除d则无解，否则$\bmod n$意义下的解有d个，可以通过对某个解不断地加n/d得到。

【接口】

vector<long long> line_mod_equation(long long a, long long b, long long n);

复杂度：$O(\log n)$

输　入：a, b, n　三个整数

输　出：所有$[0, n)$中满足$ax \equiv b(\bmod n)$的解

调用外部函数：

　扩展欧几里得：参见 1.2.2 节

【代码】

```
vector <long long> line_mod_equation(long long a,long long b,long long n){
    long long x,y;
    long long d=gcd(a,n,x,y);
    vector<long long> ans;
    ans.clear();
    if ( b%d==0 ){
        x%=n;x+=n;x%=n;
        ans.push_back(x*(b/d)%(n/d));
        for (long long i=1; i<d; ++i)
            ans.push_back((ans[0]+i*n/d)%n);
    }
    return ans;
}
```

【使用范例】

参见程序 POJ2115.CPP。

1.2.4　中国剩余定理

【任务】

求出方程组$x \equiv a_i(\bmod m_i)(0 \leqslant i < n)$的解$x$。

其中$m_1, m_2, m_3, \cdots, m_{n-1}$两两互质。

【说明】

令 $M_i = \prod_{j \ne i} m_j$。因为$(M_i, m_i) = 1$，故存在$p_i, q_i$，使$M_i p_i + m_i q_i = 1$。

令$e_i = M_i p_i$，有：

$$e_i \equiv \begin{cases} 0 \pmod{m_j}, & j \ne i \\ 1 \pmod{m_j}, & j = i \end{cases}$$

故$e_0 a_0 + e_1 a_1 + e_2 a_2 + \cdots + e_{n-1} a_{n-1}$是方程的一个解。

在$\left[0, \prod_{i=0}^{n-1} m_i\right)$中只有唯一解，将求出的解对$\prod_{i=0}^{n-1} m_i$取模即可。

【接口】

int CRT(int a[ ], int m[ ], int n);

复杂度：$O(n\log m)$，其中m和每个m_i同阶。

输　入：a, m　　第i个方程表示为$x \equiv a_i \pmod{m_i}$

　　　　n　　　方程个数

输　出：方程组在$\left[0, \prod_{i=0}^{n-1} m_i\right)$中的解

调用外部函数：

扩展欧几里得：参见 1.2.2 节

【代码】

```
int CRT( int a[], int m[], int n ){
    int M=1;
    for ( int i=0; i<n; ++i ) M*=m[i];
    int ret=0;
    for ( int i=0; i<n; ++i ){
        int x,y;
        int tm=M/m[i];
        extend_gcd(tm,m[i],x,y);
        ret=(ret+tm*x*a[i])%M;
    }
    return (ret+M)%M;
}
```

【使用范例】

参见程序 POJ1006.CPP。

1.2.5　求原根

【任务】

求一个mod p意义下的原根（p为素数）。

【说明】

原根的分布比较广，并且最小的原根通常也较小，故可以通过从小到大枚举正整数来快速地寻找一个原根。对于一个待检查的 p，对$p-1$的每一个素因子a，检查$g^{(p-1)/a} \equiv 1 \pmod p$是否成立，如果成立则说明$g$不是原根。

【接口】

long long primitive_root (long long p);

输　入：p　一个素数

输　出：p的原根

调用外部函数：

　　快速幂：参见 1.5.1 节

【代码】

```
vector<long long> a;

bool g_test(long long g, long long p) {
    for (long long i = 0; i < a.size(); i++)
        if (pow_mod(g, (p - 1) / a[i], p) == 1)
            return 0;
    return 1;
}
long long primitive_root(long long p) {
    long long tmp = p - 1;
    for (long long i = 2; i <= tmp / i; i++)
        if (tmp % i == 0) {
            a.push_back(i);
            while (tmp % i == 0)
                tmp /= i;
        }
    if (tmp != 1) {
        a.push_back(tmp);
    }
    long long g = 1;
```

```
    while (true) {
        if (g_test(g, p))
            return g;
        ++g;
    }
}
```

【使用范例】

参见程序 SGU261.CPP。

1.2.6 平方剩余

【任务】

给定a,n（n是质数），求$x^2 \equiv a \pmod n$的最小整数解x。

【说明】

先判断是否有解，然后根据剩余类特殊判断。详见程序。

【接口】

int modsqr(int a, int n);

复杂度：$O(\log^2 n)$

输　入：a,n　两个整数，n为素数

输　出：$x^2 \equiv a \pmod n$的最小整数解x，-1表示无解

调用外部函数：

　　快速幂：参见 1.5.1 节

【代码】

```
int modsqr(int a, int n) {
    int b, k, i, x;
    if (n == 2) return a % n;
    if (pow_mod(a, (n - 1) / 2, n) == 1){
        if (n % 4 == 3)
            x = pow_mod(a, (n + 1) / 4, n);
        else{
            for (b = 1; pow_mod(b, (n - 1) / 2, n) == 1; b++);
            i = (n - 1) / 2;
            k = 0;
            do{
```

```
                    i /= 2;
                    k /= 2;
                    if ((pow_mod(a, i, n) *
                        (long long)pow_mod(b, k, n) + 1) % n == 0)
                        k += (n - 1) / 2;
                }
                while(i % 2 == 0);
                x = (pow_mod(a, (i + 1) / 2, n)
                    *(long long)pow_mod(b, k / 2, n)) % n;
            }
            if (x * 2 > n)
                x = n - x;
            return x;
        }
        return -1;
    }
```

【使用范例】

参见程序 POJ1808.CPP。

1.2.7　离散对数

【任务】

给定x, n, m，求$x^y \equiv n(\bmod m)$的解（其中m是素数）。

【说明】

我们使用 giant-step baby-step 算法。令$s = \lfloor\sqrt{m}\rfloor$，则有$y = b \times s + r(0 \leqslant r < s)$，即有$x^y = x^{b\times s} \times x^r$。将所有$x^r$放入有序表中，从小到大枚举$b$，得到：$x^{b\times s} \times x^r = n$。

把x^r看成未知数解模线性方程。若解x^r能在有序表中二分查找到，则停止枚举，此时$y = b \times s + r$。

【接口】

long long discrete_log(int x, int n, int m);

复杂度：$O(\sqrt{m})$

输　入：x, n, m　三个整数，其中m是素数

输　出：$x^y \equiv n(\bmod m)$的解，-1表示无解

调用外部函数：

快速幂：参见 1.5.1 节

【代码】

```
long long discrete_log(int x, int n, int m) {
    map<long long, int> rec;
    int s = (int)(sqrt((double)m));
    for (; (long long)s * s <= m; ) s++;
    long long cur = 1;
    for (int i = 0; i < s; ++i) {
        rec[cur] = i;
        cur = cur * x % m;
    }
    long long mul = cur;
    cur = 1;
    for (int i = 0; i < s; ++i) {
        long long more = (long long)n * pow_mod(cur, m - 2, m) % m;
        if (rec.count(more)) {
            return i * s + rec[more];
        }
        cur = cur * mul % m;
    }
    return -1;
}
```

【使用范例】

参见程序 SPOJ5154.CPP。

1.2.8 *N* 次剩余

【任务】

给定N, a, p，求出$x^N \equiv a (\bmod p)$在模p意义下的所有解（其中p是素数）。

【说明】

令g为p的原根，因为p为素数，则$\phi(p) = p - 1$，所以找到原根g就可以将$\{1,2,\cdots,p-1\}$的数与$\{g^1, g^2, \cdots, g^{p-1}\}$建立一一对应关系。

令$g^y = x, g^t = a$，则有：

$$g^{y \times N} \equiv g^t (\bmod p)$$

因为p是素数，所以方程左右都不可以为0。这样就可以将这$p-1$个取值与指数建立对应关系。原问题转化为：

$$N \times y \equiv t(\mathrm{mod}\ (p-1))$$

对于y解模线性方程就可以解决。而$g^t = a$则可以用解离散对数的方法求出。

【接口】

vector <int> residue(int p, int N, int a);

复杂度：$O(\sqrt{p})$

输　入：p, N, a　三个整数，其中p是素数

输　出：方程$x^N \equiv a(\mathrm{mod}\ p)$在$[0, p-1]$中的所有解

调用外部函数：

　扩展欧几里得：参见 1.2.2 节

　求原根：参见 1.2.5 节

　离散对数：参见 1.2.7 节

　快速幂：参见 1.5.1 节

【代码】

```
vector <int> residue(int p, int N, int a) {
    int g = primitive_root(p);
    long long m = discrete_log(g, a, p);
    vector<int> ret;
    if (a == 0) {
        ret.push_back(0);
        return ret;
    }
    if (m == -1) {
        return ret;
    }
    long long A = N, B = p - 1, C = m, x, y;
    long long d = extended_gcd(A, B, x, y);
    if (C % d != 0) return ret;
    x = x * (C / d) % B;
    long long delta = B / d;
    for (int i = 0; i < d; ++i) {
        x = ((x + delta) % B + B) % B;
        ret.push_back((int)pow_mod(g, x, p));
    }
    sort(ret.begin(), ret.end());
```

```
    ret.erase(unique(ret.begin(), ret.end()), ret.end());
    return ret;
}
```

【使用范例】

参见程序 SPOJ5154.CPP。

1.3 素数与函数

1.3.1 素数筛法

【任务】

给定一个正整数N，求出$[2,N]$中的所有素数。

【说明】

数组$valid[i]$记录i是否为素数。初始所有的$valid[i]$都为true。从2开始从小到大枚举i，若$valid[i]=\text{true}$，则把从i^2开始的每一个i的倍数的$valid$赋为false。

结束之后$valid[i]=\text{true}$的就是素数。

【接口】

void getPrime(int n,int &tot,int ans[maxn])

复杂度：O(Nlog$N$),O(N)，两种实现

输　入：N　　所需素数的范围

输　出：$\&tot$　　引用，素数总数

　　　　ans　　素数表

【代码】

```
/*素数筛法 O(NlogN)*/
#define maxn 1000000

bool valid[maxn];

void getPrime(int n,int &tot,int ans[maxn]){
    tot=0;
    int i,j;
    for (i=2;i<=n;++i)valid[i]=true;
    for (i=2;i<=n;++i) if (valid[i]){
        if (n/i<i) break;
```

```
        for(j=i*i;j<=n;j+=i)valid[j]=false;
    }
    for(i=2;i<=n;++i)if(valid[i])ans[++tot]=i;
}

/*素数筛法 O(N) */
void getPrime(int n,int &tot,int ans[maxn]){
    memset(valid,true,sizeof(valid));
    for (int i=2;i<=n;i++){
        if (valid[i]){
            tot++;
            ans[tot]=i;
        }
        for (int j=1;((j<=tot) && (i*ans[j]<=n)); j++)
        {
            valid[i*ans[j]] = false;
            if (i%ans[j] == 0) break;
        }
    }
}
```

【使用范例】

参见程序 POJ 百练 3177.CPP 和 POJ 百练 3177_2.CPP。

1.3.2　素数判定

【任务】

给一个正整数N，判定N是否为素数。

【说明】

Miller-Rabin 测试：要测试N是否为素数，首先将 $N-1$分解为$2^s d$。在每次测试开始时，先随机选一个介于$[1,N-1]$的整数a，如果对所有的$r\in[0,s-1]$都满足$a^d \bmod N\neq 1$且$a^{2^r d} \bmod N\neq -1$，则N是合数。否则，N有3/4的几率为素数。为了提高测试的正确性，可以选择不同的a进行多次测试。

【接口】

bool isPrime(int N);

复杂度：$O(\log N)$

输　入：N　待测试的整数

输　出：False表示N为合数，True表示N有很大几率为素数

调用外部函数：

　　快速幂：参见 1.5.1 节

【代码】

```
bool test(int n,int a,int d){
    if(n==2)return true;
    if(n==a)return true;
    if((n&1)==0)return false;
    while(!(d&1))d=d>>1;
    int t=pow_mod(a,d,n);
    while((d!=n-1)&&(t!=1)&&(t!=n-1)){
        t=(long long)t*t%n;
        d=d<<1;
    }
    return (t==n-1 ||(d&1)==1);
}
bool isPrime(int n){
    if(n<2)return false;
    int a[]={2,3,61};        //测试集，更广的测试范围需要更大的测试集
    for(int i=0;i<=2;++i) if(!test(n,a[i],n-1)) return false;
    return true;
}
```

【使用范例】

参见程序 HDU2138.CPP。

1.3.3　质因数分解

【任务】

给一个正整数N，将N分解质因数。

【说明】

N的质因数要么是N本身(N是素数)，要么一定小于等于$\sqrt{N}$。因此可以用小于等于$\sqrt{N}$的数对N进行试除，一直到不能除为止。这时候剩下的数如果不是1，那就是N最大的质因数。

【接口】

void factor(int n,int a[maxn],int b[maxn],int &tot);

复杂度：$O(\sqrt{n})$

输　入：n　　待分解的整数

输　出：$\&tot$　不同质因数的个数

　　　　a　　$a[i]$表示第i个质因数的值

　　　　b　　$b[i]$表示第i个质因数的指数

【代码】

```
void factor(int n,int a[maxn],int b[maxn],int &tot){
    int temp,i,now;
    temp=(int)((double)sqrt(n)+1);
    tot=0;
    now=n;
    for(i=2;i<=temp;++i)if(now%i==0){
        a[++tot]=i;
        b[tot]=0;
        while(now%i==0){
            ++b[tot];
            now/=i;
        }
    }
    if(now!=1){
        a[++tot]=now;
        b[tot]=1;
    }
}
```

【使用范例】

参见程序 POJ1142.CPP。

1.3.4　欧拉函数计算

【任务】

计算N的欧拉函数$\phi(N)$。

【说明】

定义：欧拉函数$\phi(n)$，表示小于或等于n的数中与n互质的数的数目。

欧拉函数求值的方法是：

（1）$\phi(1) = 1$

（2）若n是素数p的k次幂，$\phi(n) = p^k - p^{k-1} = (p-1)p^{k-1}$

（3）若m,n互质，$\phi(mn) = \phi(m)\phi(n)$

根据欧拉函数的定义，可以推出欧拉函数的递推式：

令p为N的最小质因数，若$p^2|N$，$\phi(N) = \phi\left(\frac{N}{p}\right) \times p$；否则$\phi(N) = \phi\left(\frac{N}{p}\right) \times (p-1)$。

【接口】

void genPhi();

复杂度：$O(N\log N)$

输　出：*phi*　全局变量，存储了1~*max*中每个数的欧拉函数。

【代码】

```
const int max = 111111;

int minDiv[max], phi[max], sum[max];

void genPhi() {
    for (int i = 1; i < max; ++ i) {
        minDiv[i] = i;
    }
    for (int i = 2; i*i < max; ++ i) {
        if (minDiv[i] == i) {
            for (int j = i*i; j < max; j += i) {
                minDiv[j] = i;
            }
        }
    }
    phi[1] = 1;
    for (int i = 2; i < max; ++ i) {
        phi[i] = phi[i / minDiv[i]];
        if ((i / minDiv[i]) % minDiv[i] == 0) {
             phi[i] *= minDiv[i];
        } else {
```

```
            phi[i] *= minDiv[i] - 1;
        }
    }
}
```

【注释】

计算单个欧拉函数的值可以直接采用定义。

【使用范例】

参见程序 POJ3090. CPP。

1.3.5　Mobius 函数计算

【任务】

计算N的 Mobius 函数$\mu(N)$。

【说明】

Mobius 函数$\mu(N)$是做 Mobius 反演的时候一个很重要的系数。

Mobius 函数的定义：如果 i 的质因数分解式内有任意一个大于 1 的指数，$\mu(i) = 0$；否则 $\mu(i) = i$的质因数分解式内质数个数$\bmod 2 \times (-2) + 1$。

Mobius 函数有个很好的性质：$\sum_{d|n} \mu(d) = [n = 1]$，由此可以递推地求 Mobius 函数。

【接口】

int getMu(int n);

复杂度：$O(n\log n)$

输　入：N　一个整数

输　出：mu　全局变量，存储了$1 \sim n$中每个数的 Mobius 函数。

【代码】

```
const int n = 1 << 20;
int mu[n];

int getMu()
{
    for (int i = 1; i <= n; i++) {
        int target = i == 1 ? 1 : 0;
        int delta = target - mu[i];
```

```
        mu[i] = delta;
        for (int j = i + i; j <= n; j+=i)
          mu[j] += delta;
    }
}
```

【注释】

计算单个 Mobius 函数的值可以直接采用定义。

【使用范例】

参见程序 MOBIUS.CPP。

1.4 数 值 计 算

1.4.1 数值积分

【任务】

给定函数$f(x)$，用数值方法求积分$\int_a^b f(x)\mathrm{d}x$。

【说明】

数值方法有很多种，仅介绍 Simpson 和 Romberg 两种方法。

Simpson 方法比较简单，是以二次曲线逼近的方式取代矩形或梯形积分公式，以求得定积分的数值近似解。

Romberg 方法比较复杂一点，详细的理论推导可以参考相关数学教材的介绍，同等的计算复杂度下，精度比 Simpson 更高。

【接口】

double simpson(const T &f, double a,double b,int n);

输　入：$\&f$　被积函数

a, b　积分上下界

n　划分份数

输　出：$\int_a^b f(x)\mathrm{d}x$

double romberg(const T&f,double a,double b,double eps=1e-8);

输　入：$\&f$　被积函数

a, b　积分上下界

eps　允许误差

输　出：$\int_a^b f(x)\mathrm{d}x$

【代码】

```
template<class T>
double simpson(const T &f,double a,double b,int n){
    const double h=(b-a)/n;
    double ans=f(a)+f(b);
    for(int i=1;i<n;i+=2)ans+=4*f(a+i*h);
    for(int i=2;i<n;i+=2)ans+=2*f(a+i*h);
    return ans*h/3;
}
template<class T>
double romberg(const T &f,double a,double b,double eps=1e-8){
    vector <double>t;
    double h=b-a,last,curr;
    int k=1,i=1;
    t.push_back(h*(f(a)+f(b))/2);            // 梯形
    do{
        last=t.back();
        curr=0;
        double x=a+h/2;
        for(int j=0;j<k;++j){
            curr+=f(x);
            x+=h;
        }
        curr=(t[0]+h*curr)/2;
        double k1=4.0/3.0,k2=1.0/3.0;
        for(int j=0;j<i;j++){
            double temp=k1*curr-k2*t[j];
            t[j]=curr;
            curr=temp;
            k2/=4*k1-k2;            // 防止溢出
            k1=k2+1;
        }
        t.push_back(curr);
        k*=2;
        h/=2;
        i++;
    }while(fabs(last-curr)>eps);
    return t.back();
```

```
}
```

【使用范例】

参见程序 ROMBERG&SIMPSON.CPP。

1.4.2 高阶代数方程求根

【任务】

给定方程$a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0 = 0$，求出该方程的所有实数解。

【说明】

对于给定的n次方程，首先对其求导，求出其导函数的所有零点，那么在导函数两个相邻的零点之间，该n次方程一定是单调的，并且最多只有一个零点，利用这个性质，我们可以二分出这个零点。

而求出导函数的零点可以递归地做下去，直到$n = 1$时，可以直接返回答案。

【接口】

vector <double>equation(vector<double> coef, int n);

输入：$coef$　方程的系数，$coef[i]$表示a_i

　　　n　　方程的次数

输出：所有实数解

【代码】

```
const double EPS = 1E-12;
const double inf = 1E+12;

inline int sign(double x) {
    return x < -EPS ? -1 : x > EPS;
}
inline double get(const vector <double>&coef, double x) {
    double e = 1, s = 0;
    for (int i = 0; i < coef.size(); ++i) s += coef[i] * e, e *= x;
    return s;
}
double find(const vector <double>&coef, int n, double lo, double hi){
    double sign_lo, sign_hi;
    if ((sign_lo = sign(get(coef, lo))) == 0) return lo;
    if ((sign_hi = sign(get(coef, hi))) == 0) return hi;
```

```
    if (sign_lo * sign_hi > 0) return inf;
    for (int step = 0; step < 100 && hi - lo > EPS; ++step) {
        double m = (lo + hi) * .5;
        int sign_mid = sign(get(coef, m));
        if (sign_mid == 0) return m;
        if (sign_lo * sign_mid < 0) {
            hi = m;
        } else {
            lo = m;
        }
    }
    return (lo + hi) * .5;
}
vector <double>equation(vector<double> coef, int n) {
    vector<double> ret;
    if (n == 1) {
        if (sign(coef[1])) ret.push_back(-coef[0] / coef[1]);
        return ret;
    }
    vector<double> dcoef(n);
    for (int i = 0; i < n; ++i) dcoef[i] = coef[i + 1] * (i + 1);
    vector<double> droot = solve(dcoef, n - 1);
    droot.insert(droot.begin(), -inf);
    droot.push_back(inf);
    for (int i = 0; i + 1 < droot.size(); ++i) {
        double tmp = find(coef, n, droot[i], droot[i + 1]);
        if (tmp < inf) ret.push_back(tmp);
    }
    return ret;
}
```

【使用范例】

参见程序 TIMUS1503.CPP。

1.5　其　　他

1.5.1　快速幂

【任务】

给定a,i,n，求$a^i \bmod n$。

【说明】

最基本的方法需要i次乘法和取模运算。

较快的算法是通过：$a^i = \left(a^{\lfloor \frac{i}{2} \rfloor}\right)^2 \times a^{i \bmod 2}$，每次将指数折半来计算。

【接口】

long long pow_mod(long long a,long long i,long long n);

复杂度：$O(\log i)$

输　入：a, i, n　三个整数

输　出：$a^i \bmod n$

【代码】

```
long long pow_mod(long long a,long long i,long long n){
    if (i==0) return 1%n;
    int temp=pow_mod(a,i>>1,n);
        temp=temp*temp%n;
    if(i&1) temp=(long long)temp*a%n;
    return temp;
}
```

【使用范例】

参见程序 POW_MOD.CPP。

1.5.2　进制转换

【任务】

把一个x进制的数转换成y进制。

【说明】

先把x进制的数转换为十进制，如果x进制的数$s = \overline{s_{n-1}s_{n-2}\cdots s_0}$，则对应的十进制数为$\sum_{i=0}^{n-1} s_i \Box x^i$，再将其不断取模再倒序，转换成$y$进制数。

【接口】

string transform(int x, int y, string s);

复杂度：$O(length)$

输　入：x, y 进制数，$2 \leqslant x, y \leqslant 36$

s　x进制数，其中每一位的10~35用A~Z表示

输　出：x进制数s对应的y进制数，其中每一位的10~35用A~Z表示

【代码】

```
string transform(int x, int y, string s) {
    string res = "";
    int sum = 0;
    for (int i = 0; i < s.length(); ++i) {
        if (s[i] == '-') continue;
        if (s[i] >= '0' && s[i] <= '9') {
            sum = sum * x + s[i] - '0';
        } else {
            sum = sum * x + s[i] - 'A' + 10;
        }
    }
    while (sum) {
        char tmp = sum % y;
        sum /= y;
        if (tmp <= 9) {
            tmp += '0';
        } else {
            tmp = tmp - 10 + 'A';
        }
        res = tmp + res;
    }
    if (res.length() == 0) res = "0";
    if (s[0] == '-') res = '-' + res;
    return res;
}
```

【使用范例】

参见程序 HDU2031.CPP。

1.5.3　格雷码

【任务】

给定一个二进制的位数n，求出一个0到2^n-1的排列，使得相邻两项（包括头尾相邻）的二进制表达中只有恰好一位不同。

【说明】

Grey 序列的第i位为 i xor ($i \gg 1$)。

【接口】

vector<int> Gray_Create(int n);

复杂度：$O(2^n)$

输　入：n　二进制位数

输　出：n位的格雷码序列

【代码】

```
vector<int> Gray_Create(int n) {
    vector<int> res;
    res.clear();
    for (int i = 0; i < (1 << n); i++)
        res.push_back(i ^ (i >> 1));
    return res;
}
```

【使用范例】

参见程序 GRAY_CODE.CPP。

1.5.4 高精度整数

【任务】

完成高精度整数的加减乘除以及取模运算。

【接口】

结 构 体：*BigNumber*

成员变量：

int $d[maxl]$　$d[0]$表示当前位数

其余$d[i]$表示第i位上的数（每4位压成一个万进制数位）

构造函数：

BigNumber(string s)　从字符串s构造

成员函数：

string toString( )　输出为字符串

重载运算符：+、 − 、 × 、/、 < 、 ==

运算过程中和结果都不能包含负数。答案最长长度为$(maxl-1)\times 4$。做除法的时候余数保存在全局变量d里面。

【代码】

```
const int ten[4]={1,10,100,1000};
const int maxl=1000;                    //最大位数
struct BigNumber{
    int d[maxl];
    BigNumber (string s){
        int len=s.size();
        d[0]=(len-1)/4+1;
        int i,j,k;
        for(i=1;i<maxl;++i)d[i]=0;
        for(i=len-1;i>=0;--i){
            j=(len-i-1)/4+1;
            k=(len-i-1)%4;
            d[j]+=ten[k]*(s[i]-'0');
        }
        while(d[0]>1 && d[d[0]]==0)--d[0];
    }
    BigNumber(){
        *this=BigNumber(string("0"));
    }
    string toString(){
        string s("");
        int i,j,temp;
        for(i=3;i>=1;--i)if(d[d[0]]>=ten[i])break;
        temp=d[d[0]];
        for(j=i;j>=0;--j){
            s=s+(char)(temp/ten[j]+'0');
            temp%=ten[j];
        }
        for(i=d[0]-1;i>0;--i){
            temp=d[i];
            for(j=3;j>=0;--j){
                s=s+(char)(temp/ten[j]+'0');
                temp%=ten[j];
            }
        }
```

```
        return s;
    }
} zero("0"),d,temp,mid1[15];

bool operator <(const BigNumber &a,const BigNumber &b){
    if(a.d[0]!=b.d[0])return a.d[0]<b.d[0];
    int i;
    for(i=a.d[0];i>0;--i)if(a.d[i]!=b.d[i])return a.d[i]<b.d[i];
    return false;
}

BigNumber operator +(const BigNumber &a,const BigNumber &b){
    BigNumber c;
    c.d[0]=max(a.d[0],b.d[0]);
    int i,x=0;
    for(i=1;i<=c.d[0];++i){
        x=a.d[i]+b.d[i]+x;
        c.d[i]=x%10000;
        x/=10000;
    }
    while(x!=0){
        c.d[++c.d[0]]=x%10000;
        x/=10000;
    }
    return c;
}

BigNumber operator -(const BigNumber &a,const BigNumber &b){
    BigNumber c;
    c.d[0]=a.d[0];
    int i,x=0;
    for(i=1;i<=c.d[0];++i){
        x=10000+a.d[i]-b.d[i]+x;
        c.d[i]=x%10000;
        x=x/10000-1;
    }
    while((c.d[0]>1)&&(c.d[c.d[0]]==0))--c.d[0];
    return c;
}
```

```

BigNumber operator *(const BigNumber &a,const BigNumber &b){
    BigNumber c;
    c.d[0]=a.d[0]+b.d[0];
    int i,j,x;
    for(i=1;i<=a.d[0];++i){
        x=0;
        for(j=1;j<=b.d[0];++j){
            x=a.d[i]*b.d[j]+x+c.d[i+j-1];
            c.d[i+j-1]=x%10000;
            x/=10000;
        }
        c.d[i+b.d[0]]=x;
    }
    while((c.d[0]>1)&&(c.d[c.d[0]]==0))--c.d[0];
    return c;
}

bool smaller(const BigNumber &a,const BigNumber &b,int delta){
    if(a.d[0]+delta!=b.d[0])return a.d[0]+delta<b.d[0];
    int i;
    for(i=a.d[0];i>0;--i)if(a.d[i]!=b.d[i+delta])
        return a.d[i]<b.d[i+delta];
        return true;
}

void Minus(BigNumber &a,const BigNumber &b,int delta){
    int i,x=0;
    for(i=1;i<=a.d[0]-delta;++i){
        x=10000+a.d[i+delta]-b.d[i]+x;
        a.d[i+delta]=x%10000;
        x=x/10000-1;
    }
    while((a.d[0]>1)&&(a.d[a.d[0]]==0))--a.d[0];
}

BigNumber operator *(const BigNumber &a,const int &k){
    BigNumber c;
    c.d[0]=a.d[0];
```

```
    int i,x=0;
    for(i=1;i<=a.d[0];++i){
        x=a.d[i]*k+x;
        c.d[i]=x%10000;
        x/=10000;
    }
    while(x>0){
        c.d[++c.d[0]]=x%10000;
        x/=10000;
    }
    while((c.d[0]>1)&&(c.d[c.d[0]]==0))--c.d[0];
    return c;
}

BigNumber operator /(const BigNumber &a,const BigNumber &b){
    BigNumber c;
    d=a;
    int i,j,temp;
    mid1[0]=b;
    for(i=1;i<=13;++i){
        mid1[i]=mid1[i-1]*2;
    }
    for(i=a.d[0]-b.d[0];i>=0;--i){
        temp=8192;
        for(j=13;j>=0;--j){
            if(smaller(mid1[j],d,i)){
                Minus(d,mid1[j],i);
                c.d[i+1]+=temp;
            }
            temp/=2;
        }
    }
    c.d[0]=max(1,a.d[0]-b.d[0]+1);
    while((c.d[0]>1)&&(c.d[c.d[0]]==0))--c.d[0];
    return c;
}

bool operator ==(const BigNumber &a,const BigNumber &b){
    int i;
```

```
    if(a.d[0]!=b.d[0])return false;
    for(i=1;i<=a.d[0];++i)if(a.d[i]!=b.d[i])return false;
    return true;
}
```

【使用范例】

参见程序 POJ 百练 2980.CPP, POJ 百练 2981.CPP。

1.5.5 快速傅立叶变换

【任务】

快速实现多项式相乘或者高精度乘法。

【说明】

DFT（离散傅立叶变换）是一种对n个元素的数组的变换，根据式子直接的方法是$O(n^2)$的，但是用分治的方法可以做到$O(n\log n)$，这就是 FFT（快速傅立叶变换）。由于 DFT 变化满足 cyclic convolution 的性质，即

定义$h := a\ (*)\ b$为$h_r = \sum_{x+y=r(\mathrm{mod}\,n)} a_x b_y$，

则有$\mathrm{DFT}(a\ (*)\ b) = \mathrm{DFT}(a)\cdot\mathrm{DFT}(b)$，右边的是点乘。

所以$a\ (*)\ b = \mathrm{DFT}^{-1}(\mathrm{DFT}(a)\cdot\mathrm{DFT}(b))$，即只要对$a, b$分别进行 DFT 变化之后点乘之后再逆变换就可以了。

而实现了$a\ (*)\ b$，如果高位都是0，则就实现了高精度乘法。

这里需要注意几个问题：

（1）首先这是 cyclic 的，所以需要保证高位有足够的0。

（2）由于 FFT 本身算法的要求，n需要是二的幂次，再补0就可以。

（3）DFT 是定义在复数上的，所以有与整数之间的变换要求。

（4）DFT 变换有一个$\frac{1}{\sqrt{n}}$的因子，所以最后需要所有数除n。

（5）高精度乘法需要在多项式乘法的基础上实现下进位。

【接口】

void FFT(Complex P[ ], int n, int oper);

复杂度：$O(n\log n)$

输　入：P　　需要进行 DFT 变换的数据

　　　　n　　数据长度

oper　　$oper = 1(-1)$表示正（逆）变换

每次调用完成一次 DFT 变换。

【代码】

```
typedef long long Int64;
const int maxn = 2000000;
const double pi = acos(-1.0);

typedef complex<double> Complex;

void build(Complex _P[], Complex P[], int n, int m, int curr, int &cnt){
    if (m == n) {
        _P[curr] = P[cnt++];
    } else {
        build(_P, P, n, m * 2, curr, cnt);
        build(_P, P, n, m * 2, curr + m, cnt);
    }
}
void FFT(Complex P[], int n, int oper){
    static Complex _P[maxn];
    int cnt = 0;
    build(_P, P, n, 1, 0, cnt);
    copy(_P, _P + n, P);
    for (int d = 0; (1 << d) < n; d++) {
        int m = 1 << d;
        int m2 = m * 2;
        double p0 = pi / m * oper;
        Complex unit_p0 = Complex(cos(p0), sin(p0));
        for (int i = 0; i < n; i += m2) {
            Complex unit = 1;
            for (int j = 0; j < m; j++) {
                Complex &P1 = P[i + j + m], &P2 = P[i + j];
                Complex t = unit * P1;
                P1 = P2 - t;
                P2 = P2 + t;
                unit = unit * unit_p0;
            }
        }
    }
}
```

【注释】

默认的 Complex 用了 C++的库，比较慢，可以自己实现一个。

【使用范例】

参见程序 FFT.CPP。

1.5.6　分数类

【任务】

完成分数的加减乘除运算。

【接口】

结构体：*Fraction*

成员变量：

　　int *num*, *den*　分子，分母

构造函数：

　　Fraction(num,den)　通过分子、分母构造

重载运算符：+、 −、 ×、/、 <、 ==

【代码】

```
struct Fraction{
    long long num;
    long long den;
    Fraction(long long num = 0, long long den = 1) {
        if (den < 0) {
            num = -num;
            den = -den;
        }
        assert(den != 0);
        long long g = gcd(abs(num), den);
        this->num = num / g;
        this->den = den / g;
    }
    Fraction operator +(const Fraction &o) const {
        return Fraction(num * o.den + den * o.num, den * o.den);
    }
    Fraction operator -(const Fraction &o) const {
```

```
        return Fraction(num * o.den - den * o.num, den * o.den);
    }
    Fraction operator *(const Fraction &o) const {
        return Fraction(num * o.num, den * o.den);
    }
    Fraction operator /(const Fraction &o) const {
        return Fraction(num * o.den, den * o.num);
    }
    bool operator <(const Fraction &o) const {
        return num * o.den < den * o.num;
    }
    bool operator ==(const Fraction &o) const {
        return num * o.den == den * o.num;
    }
};
```

【使用范例】

参见程序 FRACION.CPP。

1.5.7 全排列散列

【任务】

对一个N的全排列，返回一个整数代表它在所有排列中的排名。

同样对于一个排名我们返回原排列，排名从0到$N!-1$。

【说明】

把排列看成一个多进制数。第i位的进制是$(N-i+1)!$。设$a[i]=x$，前$i-1$个有k个比x小。那么这一位应该用$(i-1-k)\times(N-i+1)!$来作为权值，最后加上所有位的权值即可。

用数求排列的时候，从高位到低位一位一位确定。用这一位的权值除以这一位对应的阶乘，若为k，那么从当前还没用过的数中找出第$k+1$小的即可。

【接口】

void intToArray(int x,int a[MAXN]);

复杂度：$O(n^2)$

输　入：x　散列值

输　出：$a[\,]$　原排列

int arrayToInt(int a[MAXN]);

复杂度：$O(n^2)$

输　入：$a[\,]$ 给定的排列

输　出：排列对应的散列值

【代码】

```
void intToArray(int x,int a[MAXN]){
    bool used[MAXN];
    int i,j,temp;
    for(i=1;i<=n;++i)used[i]=false;
    for(i=1;i<=n;++i){
        temp=x/factorial[n-i];
        for(j=1;j<=n;++j)if(!used[j]){
            if(temp==0)break;
            --temp;
        }
        a[i]=j;
        used[j]=true;
        x%=factorial[n-i];
    }
}

int arrayToInt(int a[MAXN]){
    int ans,i,j,temp;
    ans=0;
    for(i=1;i<=n;++i){
        temp=a[i]-1;
        for(j=1;j<i;++j)if(a[j]<a[i])--temp;
        ans+=factorial[n-i]*temp;
    }
    return ans;
}
```

【使用范例】

参见程序 POJ1077.CPP。

第 2 章　图　论

2.1　图的遍历及连通性

2.1.1　前向星

【任务】

以前向星方式存储一个有向图的基本信息。

【说明】

使用链表方式存储图的边。$info[i]$为节点i的边集所对应的链表的头指针，$next[j]$为第j条边的指向下一条边的指针，$to[j]$表示第j条边所指向的节点编号。即：令$addr = info[i]$，之后不断用$addr = next[addr]$即可得到链表中节点i的所有边集的编号，其中$to[addr]$表示对应边指向的节点编号。

【接口】

结构体：$graph$

成员变量：

vector < **int** > $info$	由该点出发的所有边构成的链表的表
vector < **int** > $next$	链表中下一条边在to数组中的位置
vector < **int** > to	$to[i]$表示编号为i的边指向的节点

成员函数：

graph(int n, int m);	初始化图为n个点，m条边
void add(int i, int j);	添加(i,j)之间的边

【代码】

```
struct graph {
    typedef vector<int> VI;
    VI info, next, to;
    graph(int n = 0, int m = 0) : to(0), next(0) {
```

```
        info.resize(n);
        next.reserve(m);
        to.reserve(m);
    }

    int edge_size() {                        // 返回边的数量
        return to.size();
    }
    int vertex_size(){                       // 返回值为最大点的编号+1
        return info.size();
    }
    void expand(int i) {
        if (info.size() < i + 1)
            info.resize(i + 1);
    }
    void add(int i, int j) {                 // 添加一条 i 到 j 的边
        expand(i), expand(j);
        to.push_back(j);
        next.push_back(info[i]);
        info[i] = to.size() - 1;
    }
    void del_back() {                        //删除最后一次添加的边
        int i;
        for (i = 0; i < info.size(); i++)
            if (info[i] == to.size() - 1) {
                info[i] = next.back();
                break;
            }
        to.pop_back();
        next.pop_back();
    }
    void clear() {                           // 清空图类
        info.clear();
        next.resize(0);
        to.resize(0);
    }
};
```

【使用范例】

参见程序 POJ2367.CPP。

2.1.2 割点和桥

【任务】

给定一个无向图，找出图中的割点和桥。

【说明】

我们使用三个数组来完成这个算法：

$vis[v]$记录的是节点v当前的访问状态：1表示在栈中，0表示未访问，2表示已经访问过；

$dfn[v]$记录的是节点v被访问时的深度；

$low[v]$记录的是点v可以到达的访问时间最早的祖先。

在深度遍历图的过程中，记录下每个节点的深度。对当前节点cur，以及和它相连的节点i，有两种情况：

（1）i没被访问过，这时递归访问节点i，并用i的可以到达的最早的祖先来更新cur的low值。

（2）i当前在栈中，这时说明图中有一个环，用i的深度更新cur的low值。

cur是割点的条件：cur是根且有大于一个的儿子，或者cur不是根，且cur有一个儿子v使得$low[v] \geq dfn[cur]$。

(cur, i)是桥的条件：$low[i] > dfn[cur]$。

【接口】

void cut_bridge(int cur, int father, int dep, int n);

复杂度：$O(|E| + |V|)$

输　入：	cur	当前节点
	$father$	当前节点的父亲节点
	dep	当前节点被访问时的深度
	n	图的总点数
	$edge$	全局变量，图的邻接矩阵（点从0开始编号）
输　出：	$bridge$	全局变量，$bridge[u][v]$表示边(u, v)是否是一个桥
	cut	全局变量，$cut[v]$表示节点v是否是一个割点

【代码】

```
const int V=1000;
int edge[V][V];
int bridge[V][V], cut[V];
int low[V], dfn[V], vis[V];

void cut_bridge (int cur, int father, int dep, int n){//vertex: 0 ~ n-1
    vis[cur] = 1; dfn[cur] = low[cur] = dep;
    int children = 0;
    for (int i=0; i<n; ++i) if (edge[cur][i]) {
        if (i != father && 1 == vis[i]) {
            if (dfn[i] < low[cur])
                low[cur] = dfn[i];
        }
        if (0 == vis[i]) {
            cut_bridge (i, cur, dep+1, n);
            children++;
            if (low[i] < low[cur]) low[cur] = low[i];
            if ((father==-1 && children>1) || (father!=-1 && low[i]>=
            dfn[cur]))
                cut[cur]=true;
            if(low[i]>dfn[cur]){bridge[cur][i]=bridge[i][cur]=true;}
        }
    }
    vis[cur] = 2;
}
```

【注释】

对于每个连通块取一个点x调用cut_bridge$(x,-1,0,n)$，其中n为点数。

【使用范例】

参见程序 POJ1144.CPP。

2.1.3　双连通分量

【任务】

给定一个无向图，求出它的双连通分量。

双连通分量是指图中不包含割点的连通分量。

【说明】

和求割点类似的方法类似，在对图DFS的时候记录low和dfn。由于一个点可以属于多个双连通分量，而一条边属于唯一的双连通分量，所以我们用一个边集来描述一个双连通分量。即：属于这个边集的所有边加上这些边的端点构成一个双连通分量。

每次我们发现一条树边（从父节点指向未被访问的子节点）和回边（从子节点指向父节点），就将它压入栈中。当DFS从一个点u返回到点v时，如果$low[u] \geqslant dfn[v]$，那么我们就不断地将栈顶的边弹出，直到弹出边(v,u)为止。所有弹出的边构成了一个双连通分量。

【接口】

void biconnect(int v);

复杂度：$O(|E| + |V|)$

输　入：	v	DFS 到的当前节点
	$edge$	全局变量，$edge[i]$表示从点i连出去的边
输　出：	$connect$	全局变量，表示各个双连通分量 $connect$内的每个元素为一个双连通分量，用属于这个双连通分量的点的编号组成的$vector$表示

【代码】

```
const int maxn=1010;                    //最大点数
vector<int> edge[maxn];
vector <vector <int> > connect;
int dfn[maxn], low[maxn], in_seq[maxn];
int stack[maxn], list[maxn];
int cnt, top, pop, len;
void biconnect (int v) {
    stack[++top] = v;
    dfn[v] = low[v] = pop++;
    int i, succ;
    for (i = edge[v].size() - 1; i >= 0; i--) {
        succ = edge[v][i];
        if (dfn[succ] == -1) {
            biconnect (succ);
            if (low[succ] >= dfn[v]) {
                cnt++;
                len = 0;
                do {
```

```
                    in_seq[stack[top]] = cnt;
                    list[len++] = stack[top];
                    top--;
                } while (stack[top + 1] != succ);
                in_seq[v] = cnt;
                list[len++] = v;
                vector <int> tmp(list,list+len);
                connect.push_back(tmp);
            }
            low[v] = min(low[v], low[succ]);
        } else low[v] = min(low[v], dfn[succ]);
    }
}
```

【注释】

对于每个连通块取一个点x调用biconnect(x)。

【使用范例】

参见程序 POJ2942.CPP。

2.1.4　极大强连通分量 Tarjan 算法

【任务】

给定一个有向图，找出图中的极大强连通分量，并将属于同一个强连通分量内的点染同样的颜色。

【说明】

$dfn[i]$记录的是节点i在深度优先遍历中的访问次序；

$low[i]$记录的是点i可以到达的访问时间最早的祖先；

$Stack$是记录节点的栈。

深度优先遍历整个图，一路上标记dfn并把新节点压入栈。对于一个节点i，如果它的dfn值与low值相等，说明它无法到达它的任何一个祖先。而在栈里面i与i之后的点是一定能够与i互达的（否则在之前就会被弹出栈），所以i与栈里i之后的点形成了一个极大强连通分量。这一部分可以作为一个整体弹出。

现在考虑low值的求法。这个可以根据定义来：如果点i访问一个新点j，那么j的low值i也一定能达到，可以用$low[j]$尝试更新$low[i]$；如果点i访问一个祖先k，那么则直接用$dfn[k]$尝试更新$low[i]$。

【接口】

strongly_connected_components (const vector<pair<int, int> > &edgeList, int n, vector<int> &ans);

复杂度：O(|*V*| + |*E*|)

输　入：	*&edgeList*	图中所有的边，其中边用pair<int, int>表示
	n	图中点的数目
	&ans	染色结果

【代码】

```
struct strongly_connected_components {
    vector <int>&color;
    vector <int> Stack;
    int colorCnt, curr, *instack, *dfn, *low, *info, *next, *to;

    void dfs(int x) {
        dfn[x] = low[x] = ++curr;
        Stack.push_back(x);
        instack[x] = true;
        for (int j = info[x]; j; j = next[j])
            if (!instack[to[j]]) {
                dfs(to[j]);
                low[x] = std::min(low[x], low[to[j]]);
            } else {
                if (instack[to[j]] == 1)
                    low[x] = std::min(low[x], dfn[to[j]]);
            }
        if (low[x] == dfn[x]) {
            while (Stack.back()!= x) {
                color[Stack.back()] = colorCnt;
                instack[Stack.back()] = 2;
                Stack.pop_back();
            }
            color[Stack.back()] = colorCnt++;
            instack[Stack.back()] = 2;
            Stack.pop_back();
        }
    }
    strongly_connected_components(const std::vector<std::pair<int,
```

```
    int >> &edgeList, int n, std::vector<int>&ans): color(ans) {
        color.resize(n);
        instack = new int[n];
        dfn = new int[n];
        low = new int[n];
        info = new int[n];
        next = new int[(int)edgeList.size() + 5];
        to = new int[(int)edgeList.size() + 5];
        std::fill_n(info, n, 0);
        for (size_t i = 0; i < edgeList.size(); ++i) {
            to[i + 1] = edgeList[i].second;
            next[i + 1] = info[edgeList[i].first];
            info[edgeList[i].first] = i + 1;
        }

        std::fill_n(instack, n, 0);
        colorCnt = 0;
        curr = 0;
        for (int i = 0; i < n; i++) {
            if (!instack[i]) {
                dfs(i);
            }
        }
        delete[] instack;
        delete[] dfn;
        delete[] low;
        delete[] info;
        delete[] next;
        delete[] to;
    }
};
```

【使用范例】

参见程序 POJ2186.CPP。

2.1.5　拓扑排序

【任务】

对一个有向无环图拓扑排序。

【说明】

用一个队列实现，先把入度为0的点放入队列。然后考虑不断在图中删除队列中的点，每次删除一个点会产生一些新的入度为0的点。把这些点插入队列。

【接口】

bool toposort( );

复杂度：$O(|V|+|E|)$

输　入：n　全局变量，表示点数

　　　　g　全局变量，$g[i]$表示从点i连出去的边

输　出：返回对给定的图，是否可以拓扑排序

　　　　L　全局变量，拓扑排序的结果

【代码】

```
const int maxn = 100000 + 5;
vector <int> g[maxn];
int du[maxn], n, m, L[maxn];

bool toposort(){
    memset(du, 0, sizeof(du));
    for (int i = 0; i < n; i++)
        for (int j = 0; j < g[i].size(); j++)
            du[g[i][j]]++;
    int tot = 0;
    queue <int> Q;
    for (int i = 0; i < n; i++)
        if (!du[i]) Q.push(i);
    while (!Q.empty()) {
        int x = Q.front(); Q.pop();
        L[tot++] = x;
        for (int j = 0; j < g[x].size(); j++) {
            int t = g[x][j];
            du[t]--;
            if (!du[t])
                Q.push(t);
        }
    }
    if (tot == n) return 1;
    return 0;
}
```

【使用范例】

参见程序 POJ1094.CPP。

2.1.6　2SAT

【任务】

给一组逻辑表达式，每个表达式中恰好含两个逻辑变量，运算只包含or、not，求一组方案，使得所有表达式为true。

【说明】

对于n个逻辑变量，建立$2n$个结点，分别表示每个变量为true 还是false。对于一个表达式，$x1$ or $x2$，建两条边$!x1$ −> $x2$和$!x2$ −> $x1$，含义是：若$!x1$为true，则$x2$必定为true；若$!x2$为true，则$x1$必定为true。对$!x1$ or $x2$这样的表达式可类似建边。这样可以得到一个变量间的拓扑关系图，对该图缩强连通分量，可以得到一个DAG。若存在一个变量，它的两个结点处于同一个块，则无解。否则依照拓扑逆序，对每个结点的变量依次取值，尽量取true，可以证明，这样必定得到一组可行方案。

【接口】

bool two_SAT(int n, int m, BinExp a[MAXM], int sol[MAXN]);

复杂度：$O(n + m)$

输　入：	n	逻辑变量的个数
	m	表达式的个数
	a	含有m个表达式的数组
	sol	存放方案的数组

输　出：返回是否有解，若有解则返回true并将方案存放于sol数组中

【代码】

```
//Logic Variable
struct LogVar {
    int index;
    bool pre;
    LogVar(int _index = 0, bool _pre = false):index(_index),pre(_pre){}
};
//Binary Expression
struct BinExp {
    LogVar p, q;
```

```
    BinExp(LogVar _p = LogVar(), LogVar _q = LogVar()) : p(_p), q(_q){}
};

inline int get_value(int sol[MAXN], int n, int x) {
    int r = x > n ? x - n : x;
    if (sol[r] == -1)
        return -1;
    return x > n ? !sol[r] : sol[r];
}

void dfs(int x) {
    low[x] = dfn[x] = ++id_cnt;
    s[++top] = x;
    vis[x] = true;
    for (int i = head[x], k; i; i = h[i].next)
        if (!vis[k = h[i].to]) {
            dfs(k);
            low[x] = min(low[x], low[k]);
        }
        else
            low[x] = min(low[x], dfn[k]);
    if (dfn[x] == low[x]) {
        s[top + 1] = -1;
        for (++cnt; s[top + 1] != x; --top) {
            c[cnt].push_back(s[top]);
            belong[s[top]] = cnt;
        }
    }
}
inline bool two_SAT(int n, int m, BinExp a[MAXM], int sol[MAXN]) {
    edge_tot = 0, id_cnt = 0, cnt = 0, top = 0;
    for (int i = 1; i <= 2 * n; ++i) {
        head[i] = 0;
        vis[i] = false;
        c[i].clear();
    }
    for (int i = 1; i <= n; ++i)
        sol[i] = -1;
    for (int i = 0; i < m; ++i) {
        add_edge(a[i].p.index + a[i].p.pre * n, a[i].q.index
```

```
            + !a[i].q.pre * n);
            add_edge(a[i].q.index + a[i].q.pre * n, a[i].p.index
            + !a[i].p.pre * n);
        }
        for (int i = 1; i <= 2 * n; ++i)
            if (!vis[i])
                dfs(i);
        for (int i = 1; i <= n; ++i)
            if (belong[i] == belong[i + n])
                return false;
        for (int i = 1; i <= cnt; ++i) {
            int val = 1;
            for (int j = 0; j <int(c[i].size()); ++j) {
                int cur = c[i][j];
                if (get_value(sol, n, cur) == 0)
                    val = 0;
                for (int k = head[cur]; k; k = h[k].next)
                    if (get_value(sol, n, h[k].to) == 0)
                        val = 0;
                if (val == 0)
                    break;
            }
            for (int j = 0; j <int(c[i].size()); ++j)
                if (c[i][j] > n)
                    sol[c[i][j] - n] = !val;
                else
                    sol[c[i][j]] = val;
        }
        return true;
    }
```

【使用范例】

参见程序 POI2001_PEACEFUL.CPP。

2.2 路　　径

2.2.1 Dijkstra

【任务】

用Dijkstra算法求单源最短路。图中不能有负权的边。

【说明】

Dijkstra算法按从源点src到其他各点的最短路长度递增的顺序，依次确定src到每个点的最短路。首先将$dis[src]$赋为0，其余点的dis赋为正无穷，此时所有点的最短路都还未确定。之后，每次在还未确定最短路的点中，取一个当前已得的所有可能的路径长度中最短的那个点确定，设此点为$mark$。然后对所有与$mark$相连的点进行松弛操作，即对于边$(mark, v)$，判断$dis[v]$是否大于$dis[mark]+g[mark][v]$，若是，则更新$dis[v]$为$dis[mark]+g[mark][v]$。如此做N遍后，即确定了src到所有N个点的最短距离。

【接口】

void dijkstra( );

复杂度：$O(N^2)$

输　入：N　全局变量，图中的点数

　　　　g　全局变量，$g[i][j]$表示i到j之间边的距离

输　出：dis　全局变量，$dis[i]$表示节点1到i的最短距离

【代码】

```
const int MaxN=1000;
int dis[MaxN],g[MaxN][MaxN],N;
bool v[MaxN];

void dijkstra(){
    for (int i=1; i<=N; ++i) dis[i]=INF;
    dis[1]=0;
    memset(v,0,sizeof v);
    for (int i=1; i<=N; ++i) {
        int mark=-1,mindis=INF;
        for (int j=1; j<=N; ++j)
            if (!v[j]&&dis[j]<mindis) {
                mindis=dis[j];
                mark=j;
            }
        v[mark]=1;
        for (int j=1; j<=N; ++j) if (!v[j])
            dis[j]=min(dis[j],dis[mark]+g[mark][j]);
    }
}
```

【使用范例】

参见程序 POJ1502_DIJKSTRA.CPP。

2.2.2　SPFA

【任务】

用SPFA算法求单源最短路。

【说明】

SPFA其实是Bellman-Ford的队列优化。我们用数组$dist$记录每个结点的最短路径估计值，并用邻接表来存储图g。我们采取的方法是松弛：设立一个先进先出的队列用来保存待优化的结点，优化时每次取出队首结点u，并且用u点当前的最短路径估计值对u点所指向的结点v进行松弛操作，如果v点的最短路径估计值有所调整，且v点不在当前的队列中，就将v点放入队尾。这样不断从队列中取出结点来进行松弛操作，直至队列空为止。

只要最短路径存在，上述 SPFA 算法必定能求出最小值。因为每次将点放入队尾，都是经过松弛操作达到的。换言之，每次的优化将会有某个点v的最短路径估计值$d[v]$变小。所以算法的执行会使d越来越小。由于我们假定图中不存在负权回路，所以每个结点都有最短路径值。因此，算法不会无限执行下去，随着d值的逐渐变小，直到到达最短路径值时，算法结束，这时的最短路径估计值就是对应结点的最短路径值。

【接口】

void spfa( );

复杂度：最坏情况$O(|V| \times |E|)$

输　入：	n	全局变量，图的点数
	src	全局变量，表示源点
	g	全局变量，邻接表存储所有边
		$g[i][j].first$表示i节点的第j条边的节点编号
		$g[i][j].second$表示边的长度
输　出：	$dist$	全局变量，$dist[i]$表示源点src到i的最短距离

【代码】

```
const int maxn = 1000;

int n, m, src;
vector<pair<int, int> > g[maxn + 10];
```

```

int dist[maxn + 10];
bool inQue[maxn + 10];
queue<int> que;

void spfa(){
    memset(dist, 63, sizeof(dist));
    dist[src] = 0;
    while (!que.empty()) que.pop();
    que.push(src);
    inQue[src] = true;
    while (!que.empty()) {
        int u = que.front();
        que.pop();
        for (int i = 0; i < g[u].size(); i++)
            if (dist[u] + g[u][i].second < dist[g[u][i].first]) {
                dist[g[u][i].first] = dist[u] + g[u][i].second;
                if (!inQue[g[u][i].first]) {
                    inQue[g[u][i].first] = true;
                    que.push(g[u][i].first);
                }
            }
        inQue[u] = false;
    }
}
```

【使用范例】

参见程序 POJ1502_SPFA.CPP。

2.2.3 Floyd-Warshall

【任务】

用Floyd算法求图中任意两点之间的最短距离。

【说明】

Floyd-Warshall 算法的原理是动态规划。

设$D[i][j][k]$为从i到j只以$1\sim k$中的节点为中间节点的最短路径的长度，则：

（1）若最短距离经过点k，那么$D[i][j][k] = D[i][k][k-1] + D[k][j][k-1]$

（2）若最短距离不经过点k，那么$D[i][j][k] = D[i][j][k-1]$

因此，$D[i][j][k] = \min(D[i][j][k-1], D[i][k][k-1] + D[k][j][k-1])$。

如果我们把k放在最外层的循环，那么第三维在实现上可以省去。

【接口】

void floyd( );

复杂度：$O(N^3)$

输　入：N　全局变量，图中的点数

　　　　g　全局变量，$g[i][j]$表示点i到j之间边的距离

输　出：g　全局变量，$g[i][j]$表示点i到j之间的最短距离

【代码】

```
const int MaxN=111;
const int INF=1000000000;
int N,g[MaxN][MaxN];

void floyd(){
    for (int k=1; k<=N; ++k)
        for (int i=1; i<=N; ++i)
            for (int j=1; j<=N; ++j)
                g[i][j]=min(g[i][j],g[i][k]+g[k][j]);
}
```

【使用范例】

参见程序 POJ1502_FLOYD.CPP。

2.2.4　无环图最短路

【任务】

给定一个有向无环图，求出从s到t的最短（长）路。

【说明】

首先拓扑排序，然后按照拓扑序进行动态规划即可：

$$dist[v] = \min \text{ or } \max\{dist[u] + e(u,v) | (u,v) \in E\}$$

也可以不拓扑排序，直接用记忆化搜索。

【接口】

int dag_path(int x);

复杂度：$O(n^2)$

输　入：x　起始点

n　全局变量，图中的点数

g　全局变量，邻接带权矩阵

f　全局变量，$f[i]$表示点i到终点的最短路径长度

输　出：x点到终点的最短路

【代码】

```
#define maxn 510
int g[maxn][maxn],f[maxn],n;
bool done[maxn];

int dag_path(int x){
    if(done[x])return f[x];
    for(int i=1;i<=n;++i)if(g[i][x])f[x]=max(f[x],solve(i)+g[i][x]);
    done[x]=true;
    return f[x];
}
```

【注释】

bool $done[i]$代表点i是否已经计算过。初始时$done[t]$ = true，$f[t] = 0$，其余的$done$都是false。

【使用范例】

参见程序 TIMUS1450.CPP。

2.2.5　第 *k* 短路

【任务】

求有向图中s到t的第k短路。

【说明】

先用Dijkstra算法计算出每个点i到t的最短路径长度，设为$dist[i]$，再用当前已走过的长度+$dist[i]$作为启发函数，进行 A*搜索。在 A*搜索的过程中不进行判重，而把到一个点的所有可能方案加入状态集并扩展。当第k次到达t的节点时就求出了第k短路。

具体实现要用堆才足够优化。

【接口】

int solve(vector <pair<pair<int, int>, int> >& edges, int s, int t, int k);

输入：*&edges*　　图中所有的边，其中边用pair<int, int>表示

　　　s, t, k　　s和t分别表示起点和终点，k表示要求第k短路

输出：第k短路的长度，没有第k短路则返回−1

【代码】

```
const int INF = 1000000000;
const int maxNode = 1111;
const int maxEdge = 111111;

int nodeCount, edgeCount, firstEdge[maxNode], to[maxEdge], length
    [maxEdge], nextEdge[maxEdge], dist[maxNode];
bool visit[maxNode];
priority_queue <pair <int, int> > heap;

void clearEdge() {
    nodeCount = edgeCount = 0;
    memset(firstEdge, -1, sizeof(firstEdge));
}
void addEdge(int u, int v, int w) {
    nodeCount = max(nodeCount, max(u, v));
    to[edgeCount] = v;
    length[edgeCount] = w;
    nextEdge[edgeCount] = firstEdge[u];
    firstEdge[u] = edgeCount ++;
}
int solve(vector <pair <pair <int, int>, int> >&edges,
            int s, int t, int k) {
    clearEdge();
    for (vector <pair <pair <int, int>, int> > :: iterator
            iter = edges.begin(); iter != edges.end(); ++ iter) {
        addEdge(iter->first.second, iter->first.first, iter->second);
    }
    for (int i = 1; i <= nodeCount; ++ i) {
        dist[i] = INF;
        visit[i] = false;
```

```
    }
    dist[t] = 0;
    while (1) {
        int pivot = 1;
        while (pivot <= nodeCount && visit[pivot]) {
            pivot ++;
        }
        if (pivot > nodeCount) {
            break;
        }
        for (int i = 1; i <= nodeCount; ++ i) {
            if (!visit[i] && dist[i] < dist[pivot]) {
                pivot = i;
            }
        }
        visit[pivot] = true;
        for (int iter = firstEdge[pivot];
                iter != -1; iter = nextEdge[iter]) {
            if (dist[pivot] + length[iter] < dist[to[iter]]) {
                dist[to[iter]] = dist[pivot] + length[iter];
            }
        }
    }
    clearEdge();
    for (vector <pair<pair<int, int>, int> > :: iterator
                iter = edges.begin(); iter != edges.end(); ++ iter) {
        addEdge(iter->first.first, iter->first.second, iter->second);
    }
    while (!heap.empty()) {
        heap.pop();
    }
    heap.push(make_pair(-dist[s], s));
    while (!heap.empty()) {
        pair <int, int> ret = heap.top();
        heap.pop();
        int real = -ret.first - dist[ret.second];
        if (ret.second == t) {
            if (!-- k) {
                return real;
```

```
                }
            }
            for (int iter = firstEdge[ret.second];
                    iter != -1; iter = nextEdge[iter]) {
                heap.push(make_pair(-(real + length[iter] + dist[to[iter]]),
                        to[iter]));
            }
        }
        return -1;
    }
```

【使用范例】

参见程序 POJ2449.CPP。

2.2.6　欧拉回路

【任务】

给定一个图（有向无向皆可），求一条欧拉回路的方案。

【说明】

首先来检查是否存在欧拉回路：无向图的条件是所有点度为偶数，有向图的条件是所有点出入度相同。如果有解，那么任取一个开始点。欧拉回路有一个这样的性质：如果从一个图G中去掉一个圈得到的新图G'有欧拉回路，那么G也有欧拉回路。基于这个性质，我们一旦找到一个圈就将这个圈从图里拿出来，反复如此知直到图为空。

【接口】

bool solve( );

复杂度：$O(|V| + |E|)$

输　入：adj　　全局变量，$adj[i]$表示从节点i连出的所有边

　　　　　　　　边用pair保存，pair中为<标号,相邻点>

输　出：返回是否有解。如果有解，返回true并将欧拉回路存放在$path$中。

　　　　$path$　　全局变量，欧拉回路的边顺序

【代码】

```
const int maxn  =   1995;
const int maxm  =   1000000;

```

```
int father[maxn];
vector< pair<int,int> > adj[maxn];
bool vis[maxm];

int getFather(int x)
{
    return x==father[x]?x:father[x]=getFather(father[x]);
}

void add(int x,int y,int z)
{
    adj[x].push_back(make_pair(z,y));
    adj[y].push_back(make_pair(z,x));
}

vector<int> path;

#define eid first
#define vtx second

void dfs(int u)
{
    for (int it=0;it<adj[u].size();++it)
    if (!vis[adj[u][it].eid]){
        vis[adj[u][it].eid]=true;
        dfs(adj[u][it].vtx);
        path.push_back(adj[u][it].eid);
    }
}

#undef eid
#undef vtx

bool solve()
{
    for (int i=0;i<maxn;++i) father[i]=i;
    for (int i=0;i<maxn;++i){
        for (int j=0;j<adj[i].size();++j){
            father[getFather(i)]=getFather(adj[i][j].second);
```

```
        }
    }
    int origin=-1;
    for (int i=0;i<maxn;++i)if (adj[i].size()){
        if (adj[i].size()%2==1) return false;
        if (origin==-1) origin=i;
        if (getFather(i)!=getFather(origin)) return false;
        sort(adj[i].begin(),adj[i].end());
    }

    path.clear();
    memset(vis,false,sizeof(vis));
    if (origin!=-1) dfs(origin);
    reverse(path.begin(),path.end());

    return true;
}
```

【注释】

欧拉路与欧拉回路算法基本类似，不同的地方在有解的判断和初始点的选取上，并且去圈最后结果会剩下一条路径。

【使用范例】

参见代码 POJ1041.CPP。

2.2.7　混合图欧拉回路

【任务】

给定一个无向边与有向边混合的图，判断此图是否存在一条欧拉回路。

【说明】

edge[]记录流图信息；

degree[]记录原图的出入度之差；

首先将原图中的无向边任意定向，统计每个点的出入度之差。如果有任何一个点的差值为奇数则无解。否则构造流图：原图中的每个点与流图中的点一一对应，原图中的每条无向边对应到流图中的相应位置，方向与之前的定向一致，容量为1。添加一个源一个汇，源向所有度数差为正的点连一条容量为度数差/2的边，所有度数差为负的点向汇连一条容

量为度数差/2（取绝对值）的边。易知从源出去的总容量与从汇进去的总容量相等。对此流图做最大流，如果源汇两端均流满则有解，否则无解。

【接口】

bool Work( );

输入：n　　图中的点数

　　　m　　图中的边数

输出：返回代表是否有解

【代码】

```
const int maxn = 300;
const int maxm = 100000;
const int inf = 0x7fffffff;
struct Edge {
    int data, next, cap, flow, oppo;
    Edge() {
    }
    Edge(int data, int next, int cap, int flow, int oppo) : data(data),
        next(next), cap(cap), flow(flow), oppo(oppo) {
    }
};
Edge edge[maxm];
int link[maxm][3];
int list[maxn];
int degree[maxn];
int queue[maxn], path[maxn], add[maxn];
int n, m, e, V;
void Add_Link(int a, int b, int c)
{
    edge[e] = Edge(b, list[a], c, 0, e + 1);
    edge[e + 1] = Edge(a, list[b], 0, 0, e);
    list[a] = e;
    list[b] = e + 1;
    e += 2;
}
void Init()
{
    int i;
    V = n + 2;
```

```
    for (i = 0; i < V; i++) {
        list[i] = -1;
        degree[i] = 0;
    }
    e = 0;
    for (i = 0; i < m; i++) {
        degree[link[i][0]]--;
        degree[link[i][1]]++;
        if (!link[i][2]) Add_Link(link[i][0],link[i][1], 1);
    }
}

int Max_Flow()
{
    int ans = 0, head, tail, curr, succ, i, j, k;
    bool flag = 1;
    while (flag) {
        flag = 0;
        for (i = 0; i < V; i++)
            path[i] = -1;
        path[n] = -2;
        queue[0] = n;
        add[n] = inf;
        for (head = tail = 0; !flag && head <= tail; head++) {
            curr = queue[head];
            for (i = list[curr]; i != -1; i = edge[i].next)
                if (path[succ = edge[i].data] == -1 && edge[i].flow <
                edge[i].cap) {
                    queue[++tail] = succ;
                    path[succ] = i;
                    add[succ] = min(add[curr], edge[i].cap -
                    edge[i].flow);
                    if (succ == n + 1) {
                        ans += add[succ];
                        flag = 1;
                        for(j= succ; path[j] >= 0; j = edge[k].data) {
                            k = edge[path[j]].oppo;
                            edge[path[j]].flow += add[succ];
                            edge[k].flow -= add[succ];
                        }
```

```
                    break;
                }
            }
        }
    }
    return ans;
}
bool Work(){
    init();
    int i, ans = 0;
    for (i = 0; i < n; i++)
        if (degree[i] & 1)
            return false;
    for (i = 0; i < n; i++) {
        if (degree[i] < 0) {
            Add_Link(n, i, -degree[i] / 2);
            ans += -degree[i] / 2;
        }
        if (degree[i] > 0)
            Add_Link(i, n + 1, degree[i] / 2);
    }
    if (Max_Flow() < ans) return false;
    return true;
}
```

【注释】

残量网络中蕴含了定向的信息。如果要求具体回路的方案，可以先利用残量网络对无向边进行定向，然后调用普通欧拉回路的算法。

【使用范例】

参见程序 POJ1637.CPP。

2.3 匹　配

2.3.1 匈牙利算法

【任务】

给定一个二分图，用匈牙利算法求这个二分图的最大匹配数。

【说明】

求最大匹配，那么我们希望每一个在左边的点都尽量找到右边的一个点和它匹配。我们依次枚举左边的点x的所有出边指向的点y，若y之前没有被匹配，那么(x,y)就是一对合法的匹配，我们将匹配数加一，否则我们试图给原来匹配y的x'重新找一个匹配，如果x'匹配成功，那么(x,y)就可以新增为一对合法的匹配。给x'寻找匹配的过程可以递归解决。

【接口】

int hungary( );

复杂度：$O(|E|\sqrt{|V|})$

输　入：n　　全局变量，一侧的点数

　　　　g　　全局变量，$g[i]$表示与左边点i相连的右边的点

输　出：最大匹配数

　　　　$from$　　全局变量，$mx[i]$表示最大匹配中与左边点i相连的边

【代码】

```
const int MAXN = 555;
const int n=100;

vector<int> g[MAXN];
int from[MAXN], tot;
bool use[MAXN];

bool match(int x) {
    for (int i = 0; i < g[x].size(); ++i)
    if (!use[g[x][i]]) {
        use[g[x][i]] = true;
        if (from[g[x][i]] == -1 || match(from[g[x][i]])) {
            from[g[x][i]] = x;
            return true;
        }
    }
    return false;
}

int hungary() {
    tot=0;
    memset(from, 255, sizeof from);
```

```
    for (int i = 1; i <= n; ++i) {
        memset(use, 0, sizeof use);
        if (match(i))
            ++tot;
    }
    return tot;
}
```

【使用范例】

参见程序 POJ1469_2.CPP。

2.3.2 Hopcroft-Karp 算法

【任务】

给定一个二分图，用Hopcroft-Karp算法求这个二分图的最大匹配数。

【说明】

$dx[\,], dy[\,]$分别表示二分图左右部顶点的距离标号；

$mx[\,], my[\,]$分别表示二分图左右部顶点的匹配节点。

Hopcroft相比普通的匈牙利算法来说，由于每次是增广一系列路径，所以更快。我们每次从所有未匹配的左部节点开始 BFS，进行距离标号。对于每一个队列中的左部节点X，考虑与它相邻的所有右部节点Y：如果Y是一个未匹配的右部节点，则说明至少还存在一条增广路，用一个bool变量$flag$记录，以便之后增广；否则，将Y的匹配节点加入到队列中。顺便求出距离标号。当 BFS 结束时，若不存在增广路（即$flag$为false），那么算法结束；否则对于每一个没有匹配的左部节点X执行匈牙利算法的find(X)操作。在这里，find(X)过程中，只考虑这样的边(u, v)：满足$dx[u] + 1 = dy[v]$。

【接口】

int matching( );

复杂度：$O(|E|\sqrt{|V|})$

输　入：$n1$　　全局变量，左边的点数

　　　　$n2$　　全局变量，右边的点数

　　　　g　　全局变量，$g[i]$表示与左边点i相连的右边的点

输　出：最大匹配数

　　　　mx　　全局变量，$mx[i]$表示最大匹配中与左边点i相连的边

　　　　my　　全局变量，$my[i]$最大匹配中与右边点i相连的点

【代码】

```
const int maxn = 50000;

int n1, n2;
vector<int> g[maxn + 10];
int mx[maxn + 10], my[maxn + 10];
queue<int> que;
int dx[maxn + 10], dy[maxn + 10];
bool vis[maxn + 10];

bool find(int u){
    for (int i = 0; i < g[u].size(); i++)
        if (!vis[g[u][i]] && dy[g[u][i]] == dx[u] + 1) {
            vis[g[u][i]] = true;
            if (!my[g[u][i]] || find(my[g[u][i]])) {
                mx[u] = g[u][i];
                my[g[u][i]] = u;
                return true;
            }
        }
    return false;
}

int matching(){
    memset(mx, 0, sizeof(mx));
    memset(my, 0, sizeof(my));
    int ans = 0;
    while (true) {
        bool flag = false;
        while (!que.empty()) que.pop();
        memset(dx, 0, sizeof(dx));
        memset(dy, 0, sizeof(dy));
        for (int i = 1; i <= n1; i++)
            if (!mx[i]) que.push(i);
        while (!que.empty()) {
            int u = que.front();
            que.pop();
            for (int i = 0; i < g[u].size(); i++)
```

```
                    if (!dy[g[u][i]]) {
                        dy[g[u][i]] = dx[u] + 1;
                        if (my[g[u][i]]) {
                            dx[my[g[u][i]]] = dy[g[u][i]] + 1;
                            que.push(my[g[u][i]]);
                        } else
                            flag = true;
                    }
            }
            if (!flag) break;
            memset(vis, 0, sizeof(vis));
            for (int i = 1; i <= n1; i++)
                if (!mx[i] && find(i)) ans++;
        }
        return ans;
    }
```

【使用范例】

参见程序 POJ1469.CPP。

2.3.3 KM 算法

【任务】

给定一个带权的二分图，求权值最大的完备匹配。

【说明】

$w[\,][\,]$记录边权；

$x[\,],y[\,]$分别记录两侧的点标，它随时满足$x[i]+y[j]\geqslant w[i][j]$。

KM算法基于如下事实：如果存在一个完备匹配满足$x[i]+y[j]=w[i][j]$，那么这个完备匹配就是我们要求的答案。初始时置$x[i]=\max\{w[i][j],\text{for any } j\}$，$y[i]=0$。定义所有满足$x[i]+y[j]=w[i][j]$的边组成的图为$G'$。若$G'$满足匹配存在则找到解，否则我们通过不断修改点标使得满足要求的匹配存在。

如果匹配不存在，我们将所有上次遍历过的x减去一个d，并将所有上次遍历过的y加上一个d。对于这个新点标重新取一个G'，并重复以上步骤。这个d要能刚好使得一组(x,y)进入G'，所以我们取$d=\min\{x[i]+y[j]-w[i][j]$，$i$被遍历过，$j$没遍历过$\}$。这样使得至少有一条边进入$G'$。

如果我们每次通过两层循环来决定d，那么整体复杂度是$O(n^4)$的。

我们可以通过引入松弛值$slack$来优化算法。定义$slack[i]=\min\{w[k][i]$，k被遍历过，i没遍历过$\}$，那么可以动态维护$slack[]$而不是每次重求。这样可以将复杂度降到$O(n^3)$。

【接口】

int km( );

复杂度：$O(n^3)$，n为图的点数

输　入：w　　全局变量，表示带权图

　　　　pop　　全局变量，表示图一侧的点数

输　出：最大权匹配的值

　　　　son_y　　全局变量，表示匹配方案

【代码】

```
const int maxn=555;
const int inf=1000000000;
int w[maxn][maxn],x[maxn],y[maxn];
int prev_x[maxn],prev_y[maxn],son_y[maxn],slack[maxn],par[maxn];
int lx,ly,pop;
void adjust(int v) {
    son_y[v] = prev_y[v];
    if(prev_x[son_y[v]]!=-2)
        adjust(prev_x[son_y[v]]);
}
bool find(int v) {
    int i;
    for(i=0;i<pop;i++)
        if(prev_y[i]==-1){
            if(slack[i]>x[v]+y[i]-w[v][i]){
                slack[i]=x[v]+y[i]-w[v][i];
                par[i]=v;
            }
            if(x[v]+y[i]==w[v][i]){
                prev_y[i]=v;
                if(son_y[i]==-1){
                    adjust(i);
                    return true;
                }
                if(prev_x[son_y[i]]!=-1)
                    continue;
```

```
                    prev_x[son_y[i]]=i;
                    if(find(son_y[i]))
                        return true;
                }
            }
        return false;
}
int km(){
        int i,j,m;
        for(i=0;i<pop;i++){
            son_y[i]=-1;
            y[i]=0;
        }
        for(i=0;i<pop;i++){
            x[i]=0;
            for(j=0;j<pop;j++)
                x[i]=max(x[i],w[i][j]);
        }
        bool flag;
        for(i=0;i<pop;i++){
            for(j=0;j<pop;j++){
                prev_x[j]=prev_y[j]=-1;
                slack[j]=inf;
            }
            prev_x[i]=-2;
            if(find(i))continue;
            flag=false;
            while(!flag){
                m=inf;
                for(j=0;j<pop;j++)
                    if(prev_y[j]==-1)
                        m=min(m,slack[j]);
                for(j=0;j<pop;j++){
                    if(prev_x[j]!=-1)
                        x[j]-=m;
                    if(prev_y[j]!=-1)
                        y[j]+=m;
                    else
                        slack[j]-=m;
```

```
            }
            for(j=0;j<pop;j++)
                if(prev_y[j]==-1&&!slack[j]){
                    prev_y[j]=par[j];
                    if(son_y[j]==-1){
                        adjust(j);
                        flag=true;
                        break;
                    }
                    prev_x[son_y[j]]=j;
                    if(find(son_y[j])){
                        flag=true;
                        break;
                    }
                }
        }
    }
    int ans=0;
    for(int i=0;i<pop;i++)
        ans+=w[son_y[i]][i];
    return ans;
}
```

【注释】

KM算法的运行要求是必须存在一个完备匹配，如果求一个最大权匹配（不一定完备）则把不存在的边权值赋为0即可。两侧点数不相等的时候可以添加虚拟节点。

求最小权匹配则将所有的边权取相反数即可。

【使用范例】

参见程序 URAL1076.CPP。

2.3.4　一般图最大匹配

【任务】

给出一个无向图，求最大匹配。

【说明】

不断在图中寻找路径增广，直到不存在增广路径。

在寻找路径的过程中，可能出现一个奇环，这时候把奇环收缩，成为一朵“花”，并在新图上进行增广。可以发现，每一条增广路径都可以通过把“花”展开还原回去（因为一个奇环的两段路径必然是一奇一偶，总能找到一段是满足的）。

【接口】

void matching( );

复杂度：$O(n^3)$

输　入：	n	全局变量，图的点数
	a	全局变量，图的邻接矩阵
输　出：	ans	全局变量，最大匹配数
	$match$	全局变量，$match[i]$表示和i匹配的点

【代码】

```
const int MAXN=222+10;
int n,x,y,fore,rear,cnt,ans,father[MAXN],f[MAXN],
path[MAXN],tra[MAXN],que[MAXN],match[MAXN];
bool a[MAXN][MAXN],check[MAXN],treec[MAXN],pathc[MAXN];

inline void push(int x){
    que[++rear]=x;
    check[x]=true;
    if(!treec[x]){
        tra[++cnt]=x;
        treec[x]=true;
    }
}

int root(int x){return f[x]?f[x]=root(f[x]):x;}

void clear(){
    for(int i=1,j;i<=cnt;++i){
        j=tra[i];
        check[j]=treec[j]=false;
        father[j]=0,f[j]=0;
    }
}

int lca(int u,int v){
```

```
    int len=0;
    for(;u;u=father[match[u]]){
        u=root(u);
        path[++len]=u;
        pathc[u]=true;
    }
    for(;;v=father[match[v]]){
        v=root(v);
        if(pathc[v])break;
    }
    for(int i=1;i<=len;++i)
        pathc[path[i]]=false;
    return v;
}

void reset(int u,int p){
    for(int v;root(u)!=p;){
        if(!check[v=match[u]])push(v);
        if(f[u]==0)f[u]=p;
        if(f[v]==0)f[v]=p;
        u=father[v];
        if(root(u)!=p)father[u]=v;
    }
}

void flower(int u,int v){
    int p=lca(u,v);
    if(root(u)!=p)father[u]=v;
    if(root(v)!=p)father[v]=u;
    reset(u,p),reset(v,p);
}

bool find(int x){
    fore=rear=cnt=0,push(x);
    while(fore++<rear){
        int i=que[fore];
        for(int j=1;j<=n;++j)
            if(a[i][j]&&root(i)!=root(j)&&match[j]!=i)
                if(match[j]&&father[match[j]])
```

```
                    flower(i,j);
                else if(father[j]==0){
                    father[j]=i;
                    tra[++cnt]=j;
                    treec[j]=true;
                    if(match[j])
                        push(match[j]);
                    else{
                        for(int k=i,l=j,p;k;l=p,k=father[l]){
                            p=match[k];
                            match[k]=l;
                            match[l]=k;
                        }
                        return true;
                    }
                }
    }
    return false;
}

void matching()
{
    for(int i=1;i<=n;++i)
        if(match[i]==0){
            if(find(i))++ans;
            clear();
        }
}
```

【使用范例】

参见程序 TIMUS1099.CPP。

2.4 树

2.4.1 LCA

【任务】

给定一棵树，求出节点u和v的 LCA。

【说明】

对于每个节点v，记录$anc[v][k]$，表示从它向上走2^k步之后到达的节点（如果越过了根节点，那么$anc[v][k]$就是根节点。

dfs函数对树进行dfs，先求出$anc[v][0]$，再利用$anc[v][k] = anc[anc[v][k-1]][k-1]$求出其他$ans[v][k]$的值。

swim(x,k)函数从节点x向上移动k步，并将x赋为新走到的节点。

find(x,y)函数寻找x和y的 LCA。首先利用swim，将x,y调整到同一高度。如果此时x和y重合，那么这就是我们要找的 LCA。如果它们不重合，就不断地寻找一个最小的k，使得$anc[x][k] = anc[y][k]$（这说明向上走2^k步越过了x,y的 LCA），然后x,y同时向上移动2^{k-1}步，显然新的x,y和原来的x,y有相同的 LCA。直到$k = 0$，这说明此时x,y的父节点$anc[x][0]$和$anc[y][0]$重合，并且就是我们要寻找的 LCA。

【接口】

void lca(int root);

复杂度：O(N)

输　入：$root$　　树的根节点

　　　　$head$　　全局变量，存储边的信息，$head[i]$表示第i个节点的头指针

　　　　$point$　　全局变量，$point[i]$表示第i条边指向的节点

　　　　$next$　　全局变量，$next[i]$表示第i条边的下一个指针

输　出：anc　　全局变量，$anc[v][k]$表示结点v向上走2^k步之后到达的节点

int find(int x, int y);

复杂度：O(log$N$)

输　入：x,y　　询问x和y的 LCA

输　出：点x和y的 LCA

【代码】

```
void dfs(int root) {
    static int Stack[maxn];
    int top = 0;
    dep[root] = 1;
    for (int i = 0; i < maxh; i++)
        anc[root][i] = root;
    Stack[++top] = root;
    memcpy(head, info, sizeof(head));
    while (top) {
```

```
            int x = Stack[top];
            if (x != root) {
                for (int i = 1; i < maxh; i++) {
                    int y = anc[x][i - 1];
                    anc[x][i] = anc[y][i - 1];
                }
            }
            for (int &i = head[x]; i; i = next[i]) {
                int y = point[i];
                if (y != anc[x][0]) {
                    dep[y] = dep[x] + 1;
                    anc[y][0] = x;
                    Stack[++top] = y;
                }
            }
        while (top && head[Stack[top]] == 0) top--;
        }
}
void swim(int &x, int H) {
    for (int i = 0; H > 0; i++) {
        if (H & 1) x = anc[x][i];
        H /= 2;
    }
}
int lca(int x, int y) {
    int i;
    if (dep[x] > dep[y]) swap(x, y);
    swim(y, dep[y] - dep[x]);
    if (x == y) return x;
    for (;;) {
        for (i = 0; anc[x][i] != anc[y][i]; i++);
        if (i == 0) {
            return anc[x][0];
        }
        x = anc[x][i - 1];
        y = anc[y][i - 1];
    }
    return -1;
}
```

【使用范例】

参见程序 POJ3728.CPP。

2.4.2　最小生成树 Prim 算法

【任务】

Prim算法求最小（最大）生成树

【说明】

先任意找一个点标记，然后每次找一条最短的两端分别为标记和未标记的边加进来，把未标记的点标记上。即每次加入一条合法的最短的边，每次扩展一个点由未标记为已标记，直至扩展至N个点。

【接口】

int Prim( );

复杂度：O($|V|^2$)

输　入：g　　全局变量，$g[i]$表示所有与节点i相连的边

$g[i][j].first$表示与节点i的第j条边相连的节点编号

$g[i][j].second$表示距离

输　出：最小生成树的边权和

【代码】

```
void Prim(){
    memset(v,0,sizeof v);
    for (int i=1; i<=N; ++i) dis[i]=INF;
    dis[1]=0;
    int ans=0;
    for (int i=1; i<=N; ++i)
    {
        int mark=-1;
        for (int j=1; j<=N; ++j) if (!v[j])
        if (mark==-1) mark=j;
        else if (dis[j]<dis[mark]) mark=j;
        if (mark==-1) break;
        v[mark]=1;
        ans+=dis[mark];
        for (int j=0; j<g[mark].size(); ++j) if (!v[g[mark][j].first]){
```

```
            int x=g[mark][j].first;
            dis[x]=min(dis[x],g[mark][j].second);
        }
    }
    return ans;
}
```

【注释】

bool $v[i]$标记节点i是否已经加入最小生成树；

int $dis[i]$记录未标记的节点i加入最小生成树的最小权值。

【使用范例】

参见程序 POJ2395_PRIM.CPP。

2.4.3 最小生成树 Kruskal 算法

【任务】

给出带权无向图，用Kruskal算法求出其权值和最小的生成树。

【说明】

Kruskal是通过一个贪心的想法：每次取剩下的边权最小的边，如果加上这条边以后图中出现了一个环（这个可以通过并查集维护），则破坏了生成树的性质，就不选这条边。依次进行直到整张图出现一棵生成树为止。

【接口】

int kruscal( );

复杂度：$O(M\log M)$

输　入：N, M　　全局变量，图中的点数和边数

　　　　e　　全局变量，$e[i]$表示第i条边的信息（连接x与y，权值为w）

输　出：最小生成树的边权和

【代码】

```
struct edge{
    int x,y,w;
    edge(int x=0, int y=0, int w=0):x(x),y(y),w(w){}
} e[MaxM];

int getfather(int x){
```

```
    if (x==fa[x]) return x;
    else return fa[x]=getfather(fa[x]);
}
int kruscal(){
    sort(e+1,e+M+1,cmp);                           //对边按从小到大排序
    int cnt=N;
    for (int i=1; i<=N; ++i) fa[i]=i;              //初始化并查集
    for (int i=1; i<=M; ++i){
        int t1=getfather(e[i].x);
        int t2=getfather(e[i].y);
        if (t1!=t2){
            fa[t1]=t2;
            ans+=e[i].w;
            if (cnt==1) break;
            //若只剩一个联通块,即最小生成树已经得出,则退出
        }
    }
    return ans;
}
```

【注释】

$fa[i]$为i所在连通块的代表。

【使用范例】

参见程序 POJ2395_KRUSKAL.CPP。

2.4.4　单度限制最小生成树

【任务】

给出无向图G和限制L，求一个最小生成树，满足0号顶点的度恰好为L。

【说明】

对除0号顶点外的点集，求一次最小生成森林，对于最小生成森林的连通分量，选择最短的一条边与0号点连通。设此时0号点的度是k_0，如果$k_0 > L$则无解。

下面通过可行交换来增加0号点的度，即每次尝试加入一条和 0 号点相接的边，然后删去所形成环上的最长边。剩下的问题是询问每个点到根路径的最长边。从原理上讲，这个可以用动态树维护，如果时限没有太严格，可以用一个树形动态规划处理。之后每次选择增量最小的边交换，直到k_0达到L则结束。

【接口】

vector <int> restricted_mst(int n, int limit, vector <pair <pair <int, int>, int> > edges);

复杂度：$O(nlogn + limit \times n)$

输　入：n　　点数

$limit$　　度数限制

$\&edges$　　无向图的边集

输　出：返回生成树的边集，无解则返回{-1}

【代码】

```
bool compare(int i, int j) {
    return c[i] < c[j];
}

int findRoot(int i) {
    if (parent[i] != i) {
        parent[i] = findRoot(parent[i]);
    }
    return parent[i];
}

void addEdge(int i, int u, int v) {
    to[i] = v;
    nextEdge[i] = firstEdge[u];
    firstEdge[u] = i;
}

void dfs(int p, int u) {
    for (int iter = firstEdge[u]; iter != -1; iter = nextEdge[iter]) {
        if (!choose[iter >> 1]) {
            continue;
        }
        int v = to[iter];
        if (p == v) {
            continue;
        }
        if (u) {
            best[v] = c[iter >> 1];
            candidate[v] = iter >> 1;
```

```
            if (best[u] > best[v]) {
                best[v] = best[u];
                candidate[v] = candidate[u];
            }
        } else {
            best[v] = -INF;
        }
        dfs(u, v);
    }
}

void myAddEdge(int i) {
    addEdge(i + i, a[i], b[i]);
    addEdge(i + i + 1, b[i], a[i]);
}

vector <int> restricted_mst(int n, int limit, vector <pair <pair <int,
int>, int> >& edges) {
    int m = (int)edges.size();
    for (int i = 0; i < m; ++i) {
        a[i] = edges[i].first.first;
        b[i] = edges[i].first.second;
        c[i] = edges[i].second;
        if (a[i] > b[i]) {
            swap(a[i], b[i]);
        }
        order[i] = i;
    }
    sort(order, order + m, compare);
    for (int i = 0; i < n; ++i) {
        parent[i] = i;
    }
    memset(choose, 0, sizeof(choose));
    for (int i = 0; i < m; ++i) {
        int e = order[i];
        if (!a[e] || findRoot(a[e]) == findRoot(b[e])) {
            continue;
        }
        choose[e] = true;
```

```
        parent[findRoot(a[e])] = findRoot(b[e]);
    }
    int component = 0;
    for (int i = 1; i < n; ++i) {
        if (findRoot(i) == i) {
            component++;
            best[i] = INF;
        }
    }
    if (component > limit) {
        return vector <int>(1, -1);
    }
    memset(adj, -1, sizeof(adj));
    for (int i = 0; i < m; ++i) {
        if (a[i]) {
            continue;
        }
        adj[b[i]] = i;
        int r = findRoot(b[i]);
        if (c[i] < best[r]) {
            best[r] = c[i];
            candidate[r] = i;
        }
    }
    for (int i = 1; i < n; ++i) {
        if (findRoot(i) == i) {
            if (best[i] == INF) {
                return vector <int>(1, -1);
            }
            choose[candidate[i]] = true;
        }
    }
    memset(firstEdge, -1, sizeof(firstEdge));
    for (int i = 0; i < m; ++i) {
    if (choose[i]) {
            myAddEdge(i);
        }
    }
    while (component < limit) {
```

```
        dfs(-1, 0);
        int tmpBest = INF,
            tmpCandidate;
        for (int i = 1; i < n; ++i) {
            if (adj[i] == -1 || best[i] == -INF) {
                continue;
            }
            if (c[adj[i]] - best[i] < tmpBest) {
                tmpBest = c[adj[i]] - best[i];
                tmpCandidate = i;
            }
        }
        if (tmpBest == INF) {
            return vector <int>(1, -1);
        }
        choose[candidate[tmpCandidate]] = false;
        choose[adj[tmpCandidate]] = true;
        myAddEdge(adj[tmpCandidate]);
        component++;
    }
    vector <int> result;
    for (int i = 0; i < m; ++ i) {
        if (choose[i]) {
            result.push_back(i);
        }
    }
    return result;
}
```

【使用范例】

参见程序 CODEFORCES125E.CPP。

2.4.5 最小树形图

【任务】

给定一个有向图，求以某个给定顶点为根的有向生成树（也就是说沿着这$N-1$条有向边可以从根走到任意点），使权和最小。

【说明】

首先判定是否存在最小树形图，以根为起点DFS一遍即可。

（1）除根节点外，对于其他所有顶点V_i，找到一条以V_i为终点的边，加入最短弧集合；

（2）检查最短弧集合中的边是否形成有向圈，有跳至步骤3，否则跳至步骤4；

（3）将有向环缩成一个点。新图的边权更新过程见combine()函数；

（4）sum +最短弧集合中的边权和即为最小树形图的答案。

【接口】

double mdst(int root);

复杂度：$O(n^3)$

输　入：n, m　　全局变量，图的点数和边数

　　　　$root$　　给定的根

　　　　g　　全局变量，$g[i][j]$表示i到j的有向边的边权

输　出：返回最小树形图的边权和

【代码】

```
double g[maxn][maxn];
int used[maxn], pass[maxn], eg[maxn], more, queue[maxn], n, m;

inline void combine(int id, double &sum) {
    int tot = 0, from, i, j, k;
    for (; id != 0 && !pass[id]; id = eg[id]) {
        queue[tot++] = id;
        pass[id] = 1;
    }
    for (from = 0; from < tot && queue[from] != id; ++from);
    if (from == tot) return;
    more = 1;
    for (i = from; i < tot; ++i) {
        sum += g[eg[queue[i]]][queue[i]];
        if (i != from) {
            used[queue[i]] = 1;
            for (j = 1; j <= n; ++j) if (!used[j]) {
                if (g[queue[i]][j] < g[id][j])
                    g[id][j] = g[queue[i]][j];
            }
        }
    }
```

```
    for (i = 1; i <= n; ++i) if (!used[i] && i != id) {
        for (j = from; j < tot; ++j) {
            k = queue[j];
            if (g[i][id] > g[i][k] - g[eg[k]][k])
                g[i][id] = g[i][k] - g[eg[k]][k];
        }
    }
}

inline double mdst(int root) {
    int i, j, k;
    double sum = 0;
    memset(used, 0, sizeof(used));
    for (more = 1; more; ) {
        more = 0;
        memset(eg, 0, sizeof(eg));
        for (i = 1; i <= n; ++i) if (!used[i] && i != root) {
            for (j = 1, k = 0; j <= n; ++j) if (!used[j] && i != j) {
                if (k == 0 || g[j][i] < g[k][i])
                    k = j;
            }
            eg[i] = k;
        }
        memset(pass, 0, sizeof(pass));
        for (i = 1; i <= n; ++i) if (!used[i] && !pass[i] && i != root)
            combine(i, sum);
    }
    for (i = 1; i <= n; ++i) if (!used[i] && i != root) sum += g[eg[i]][i];
    return sum;
}
```

【使用范例】

参见程序 AIZU2309.CPP。

2.4.6　最优比例生成树

【任务】

给定一些边，每条边有两个权w_i和u_i。求其中$\frac{\sum w_i}{\sum u_i}$最小（最大）的一棵生成树。

【说明】

下面以求答案最小为例。

我们二分答案，假设最小的答案为$best$，我们二分的答案为ans。那么我们将每条边的边权变为$w_i - u_i \times ans$。则：

$ans < best$时，求最小生成树得到的答案> 0；

$ans = best$时，求最小生成树得到的答案$= 0$；

$ans > best$时，求最小生成树得到的答案< 0。

【接口】

double ratio_mst( );

复杂度：$O(|V|^2)$

输　入：n　全局变量，表示图的大小

$g1$　全局变量，表示u的邻接矩阵

$g2$　全局变量，表示w的邻接矩阵

输　出：最优的比例

【代码】

```
const int maxn=1001;
int n;
bool done[maxn];
double g1[maxn][maxn],g2[maxn][maxn],d[maxn];

double check(double data){
    int i,j,tj;
    double temp,ans;
    for(i=2;i<=n;++i){
        done[i]=false;
        d[i]=g2[1][i]-g1[1][i]*data;
    }
    ans=0;
    done[1]=true;
    d[1]=0;
    for(i=1;i<n;++i){
        temp=1e30;tj=0;
        for(j=2;j<=n;++j)if(!done[j] && d[j]<temp){
            tj=j;
            temp=d[j];
```

```
        }
        ans+=d[tj];
        done[tj]=true;
        for(j=2;j<=n;++j)if(!done[j])d[j]=min(d[j],g2[tj][j]-
        g1[tj][j]*data);
    }
    return ans;
}
double ratio_mst(){
    double small,mid,big;
    small=0;
    big=1e6;
    for(int i=1;i<=50;++i){
        mid=(big+small)/2;
        if(check(mid)<0) big=mid;
        else small=mid;
    }
    return (small+big)/2;
}
```

【注释】

此类二分的方法可以推广到很多问题上，如“最优比率的割”等问题。类似地还存在时间复杂度稍好一些的迭代求解的方法。

【使用范例】

参见程序 POJ2728.CPP。

2.4.7　树的直径

【任务】

在一棵树上找出相距最远的两点间的距离。连接这样两点的路径称为树的直径。

【说明】

首先任选一点，通过一遍dfs找到一个距它最远的点u，再从u开始再做一遍dfs找到距u最远的一点v。$u-v$这条路径一定是树的一个直径。

【接口】

int getDiameter(int nodeCount, vector <pair <pair <int, int>, int> > edges);

复杂度：$O(|V| + |E|)$

输　入：*nodeCount*　树的点数

　　　　edges　$<<u,v>,w>$表示边(u,v)，长度为w

输　出：树的直径

【代码】

```
const int N = 222222;

int edgeCount, firstEdge[N], to[N], length[N], nextEdge[N];
vector <int> dist;

void addEdge(int u, int v, int w) {
    to[edgeCount] = v;
    length[edgeCount] = w;
    nextEdge[edgeCount] = firstEdge[u];
    firstEdge[u] = edgeCount++;
}

void dfs(int p, int u, int d) {
    dist[u] = d;
    for (int iter = firstEdge[u]; iter != -1; iter = nextEdge[iter]) {
        if (to[iter] != p) {
            dfs(u, to[iter], d + length[iter]);
        }
    }
}

int getDiameter(int nodeCount,
        vector <pair <pair <int, int>, int> > edges) {
    edgeCount = 0;
    memset(firstEdge, -1, sizeof(firstEdge));
    for (vector <pair <pair <int, int>, int> > :: iterator
            iter = edges.begin(); iter != edges.end(); ++iter) {
        addEdge(iter->first.first, iter->first.second, iter->second);
        addEdge(iter->first.second, iter->first.first, iter->second);
    }
    dist.resize(nodeCount);
    dfs(-1, 0, 0);
    int u = max_element(dist.begin(), dist.end()) - dist.begin();
```

```
    dfs(-1, u, 0);
    return *max_element(dist.begin(), dist.end());
}
```

【使用范例】

参见程序 POJ1985.CPP。

2.5　网　络　流

2.5.1　最大流 Dinic 算法

【任务】

用Dinic算法求最大流。

【说明】

Dinic算法不断重复以下过程：

首先从源点沿着可增广边做一遍广搜，给每一个点标记一个距离。如果遍历不到汇点，即找不到增广路，算法结束。

在增广的时候，只选择距离恰好是自己距离加一的点扩展。这样保证了每次以最短路增广。其次在找到了一条增广路后，并不是立刻回退到源点，而是寻找到增广路上第一个满流的边的起点继续增广。

【接口】

int maxflow( );

复杂度：上界为$O(N^2M)$，一般效率很高

输　入：$src, sink$　　表示源点和汇点

　　　　g, e　　全局变量，表示存边的邻接表

输　出：最大流

【代码】

```
const int inf = 1000000000;
const int maxn = 20000, maxm = 500000;          //最大的点数和边数

struct Edge
{
    int v, f, nxt;
};

```

```
int n, src, sink;
int g[maxn + 10];
int nume;
Edge e[maxm * 2 + 10];

void addedge(int u, int v, int c)
{
    e[++nume].v = v;
    e[nume].f = c;
    e[nume].nxt = g[u];
    g[u] = nume;
    e[++nume].v = u;
    e[nume].f = 0;
    e[nume].nxt = g[v];
    g[v] = nume;
}

void init()
{
    memset(g, 0, sizeof(g));
    nume = 1;
    // 加边……
}

queue<int> que;
bool vis[maxn + 10];
int dist[maxn + 10];

void bfs()
{
    memset(dist, 0, sizeof(dist));
    while (!que.empty()) que.pop();
    vis[src] = true;
    que.push(src);
    while (!que.empty()) {
        int u = que.front();
        que.pop();
        for (int i = g[u]; i; i = e[i].nxt)
            if (e[i].f && !vis[e[i].v]) {
```

```
                que.push(e[i].v);
                dist[e[i].v] = dist[u] + 1;
                vis[e[i].v] = true;
            }
    }
}

int dfs(int u, int delta)
{
    if (u == sink) {
        return delta;
    } else {
        int ret = 0;
        for (int i = g[u]; delta && i; i = e[i].nxt)
            if (e[i].f && dist[e[i].v] == dist[u] + 1) {
                int dd = dfs(e[i].v, min(e[i].f, delta));
                e[i].f -= dd;
                e[i ^ 1].f += dd;
                delta -= dd;
                ret += dd;
            }
        return ret;
    }
}

int maxflow()
{
    int ret = 0;
    while (true) {
        memset(vis, 0, sizeof(vis));
        bfs();
        if (!vis[sink]) return ret;
        ret += dfs(src, inf);
    }
}
```

【注释】

addedge是加边操作，表示加一条u到v容量为c的边。请务必在开始加边之前把$nume$赋值成1，并把g数组赋值为0。

【使用范例】

参见程序 POJ1273.CPP。

2.5.2 最小割

【任务】

找出流网络$G = (V, E)$的一组最小割的边集。

【说明】

根据最大流最小割定理，我们可以知道：如果“一条边满流”和“去掉该边后网络的最大流减小的量等于该边的容量”两个条件同时满足，那么这条边一定属于最小割的边集。

因此，我们可以从S开始BFS出一个源集合，这个集合中的所有点与源连通，那么如果一条边(u, v)满流并且u属于源集合，v不属于源集合，则这条边一定属于最小割。

【接口】

vector<pair<int, int> > min_cut(int f[maxn][maxn], int c[maxn][maxn], int s, int n);

复杂度：取决于网络流算法

输　入：f　$f[i][j]$表示i到j的流量

　　　　c　$c[i][j]$表示i到j的容量

　　　　s, n　表示源和点数

输　出：一组最小割边集

【代码】

```
vector<pair<int, int> >min_cut(int f[maxn][maxn], int c[maxn][maxn],
int s, int n) {
    static bool v[maxn];
    queue<int> q;
    q.push(s);
    v[s] = true;
    for (; !q.empty(); ) {
        int x = q.front(); q.pop();
        for (int i = 1; i <= n; ++i) {
            if (f[x][i] < c[x][i] && !v[i]) {
                v[i] = true;
                q.push(i);
            }
```

```
        }
    }
    vector<pair<int, int> > res;
    for (int i = 1; i <= n; ++i) if (v[i]) {
        for (int j = 1; j <= n; ++j) if (!v[j] && f[i][j] == c[i][j] &&
        c[i][j] > 0) {
            res.push_back(make_pair(i, j));
        }
    }
    return res;
}
```

【使用范例】

参见程序 POJ2125.CPP。

2.5.3　无向图最小割

【任务】

用Stoer-Wagner算法求无向图最小割。

【说明】

定理：对于图中任意两点s和t来说，无向图G的最小割要么为s到t的割，要么是生成图$G/\{s,t\}$的割（意思是把s和t合并）。

那么算法的主步骤就是求出当前图中某两点的最小割，并将这两点合并。

快速求当前图某两点的最小割的方式：

（1）维护一个集合A，初始里面只有v_1（可以任意）这个点；

（2）取一个最大的$w(A,y)$的点y放进A集合（集合到点的权值为集合内所有点到该点的权值和）；

（3）反复2步骤，直到A集和G集相等；

（4）设最后两个添加的点为s和t，那么$w(G-\{t\},t)$的值就是s到t的cut值。

【接口】

结构体：$Stoer_Wagner$

成员变量：

int n	点数
int g[maxn][maxn]	$g[i][j]$表示i和j两点之间的最大流量

int b[maxn], $dist$[maxn]

成员函数：

void init(int nn, int w[maxn][maxn]); 初始化图，nn, w分别对应n, g

int Min_Cut( );

复杂度：$O(n^3)$

输　出：无向图最小割

【代码】

```
struct Stoer_Wagner{
    int n, g[maxn][maxn], b[maxn], dist[maxn];
    void init(int nn, int w[maxn][maxn]){
        int i, j;
        n=nn;
        for (i=1; i<=n; ++i)
            for (j=1; j<=n; ++j)
                g[i][j]=w[i][j];
    }
    int Min_Cut_Phase(int ph, int & x, int & y){
        int i, j, t;
        b[t=1]=ph;
        for (i=1; i<=n; ++i)
            if (b[i]!=ph) dist[i]=g[1][i];
        for (i=1; i<n; ++i) {
            x=t;
            for (t=0, j=1; j<=n; ++j)
                if (b[j]!=ph && (!t || dist[j]>dist[t])) t=j;
            b[t]=ph;
            for (j=1; j<=n; ++j)
                if (b[j]!=ph) dist[j]+=g[t][j];
        }
        return y=t, dist[t];
    }
    void Merge(int x, int y){
        int i;
        if (x>y) swap(x, y);
        for (i=1; i<=n; ++i)
            if (i!=x && i!=y)
                g[i][x]+=g[i][y], g[x][i]+=g[i][y];
```

```
        if (y==n) return;
        for (i=1; i<n; ++i) if (i!=y) {
            swap(g[i][y], g[i][n]);
            swap(g[y][i], g[n][i]);
        }
    }
    int Min_Cut(){
        int i, ret = 0x3fffffff, x, y;
        memset(b, 0, sizeof(b));
        for (i=1; n>1; ++i, --n) {
            ret=min(ret, Min_Cut_Phase(i, x, y));
            Merge(x, y);
        }
        return ret;
    }
}
```

【注释】

可以利用优先队列将复杂度优化到$O(nm+n^2\log n)$。

【使用范例】

参见程序 POJ2914.CPP。

2.5.4　有上下界的网络流

【任务】

求有上下界的网络流。

【说明】

设原来的源点是$Source$，汇点是$Sink$。新建一个超级源$SuperSource$和超级汇$SuperSink$。对于原网络中的每一条边$u\to v$，上界U，下界L，将它拆成三条边：

（1）$u\to SuperSink$，容量为L。

（2）$SuperSource\to v$，容量为L。

（3）$u\to v$，容量为$U-L$。

最后添加边$Sink\to Source$，容量为$+\infty$。在新建的网络上，计算从$SuperSource$到$SuperSink$的最大流。如果每条从$SuperSource$发出的边都满流，说明存在可行流，否则不存在可行流。

求出可行流之后，如果要继续求最大流，那么将这个可行流还原到原来的网络中，再从$Source$到$Sink$不断增广，直到找不到增广路为止。

如果要求最小流，可以采取下面的办法：先不连$Sink \rightarrow Source$，计算$SuperSource$到$SuperSink$的最大流。然后连$Sink \rightarrow Source$，容量$+\infty$，并不断从$SuperSource$寻找到$SuperSink$的增广路，这一步增广的总流量就是最小流。

实现的时候，要把从$SuperSource$连向同一个节点的多条边合并成一条（容量相加），提高算法效率。从同一个节点指向$SuperSink$的多条边也应合并。

【接口】

bool lowbound_flow(int n, int source, int sink,vector<int> u, vector<int> v, vector<int> L, vector<int> U);

复杂度：取决于网络流算法

输　入：	$n, source, sink$	原网络的点数、源点和汇点
	u, v	描述网络中的边：$u[i] \rightarrow v[i]$
	L, U	图中每条边的下界$L[i]$，上界$U[i]$

输　出：是否存在可行流

调用外部函数：

最大流：参见 2.5.1 节

【代码】

```
bool lowbound_flow(int n, int source, int sink,
        vector<int> u, vector<int> v, vector<int> L, vector<int> U) {
    dinic::init();
    vector<int> tot_in(n + 1), tot_out(n + 1);
    for (int i = 0; i < (int)u.size(); ++i) {
        if (U[i] < L[i]) {
            return 0;
        }
        tot_in[v[i]] += L[i];
        tot_out[u[i]] += L[i];
        dinic::addedge(u[i], v[i], U[i] - L[i]);
    }
    dinic::addedge(sink, source, 1000000000);
    int super_source = n + 1;
    int super_sink = n + 2;
    dinic::src = super_source;
    dinic::sink = super_sink;
```

```
    for (int i = 1; i <= n; ++i) {
        dinic::addedge(super_source, i, tot_in[i]);
        dinic::addedge(i, super_sink, tot_out[i]);
    }
    int ans = dinic::maxflow();
    for (int i = dinic::g[super_source]; i; i = dinic::e[i].nxt) {
        if (dinic::e[i].f != 0) {
            return 0;
        }
    }
    return 1;
}
```

【使用范例】

参见程序 POJ2396.CPP。

2.5.5　费用流

【任务】

求最小费用最大流。

【说明】

不断用spfa找增广路然后增广。spfa的距离指标就是费用。

【接口】

int mincostflow( );

输入：$src, sink$　　全局变量，表示源点和汇点

　　　g, e　　　　全局变量，表示存边的邻接表

输出：最小的费用

【代码】

```
const int maxn = 5000, maxm = 50000, inf = 1000000000;

struct Edge
{
    Edge() {};
    Edge(int a, int b, int c, int d) {v = a; f = b; w = c; nxt = d;};
    int v, f, w, nxt;
```

```
};

int n, lmt;
int g[maxn + 10];
Edge e[maxm + 10];
int nume;
int src, sink;

void addedge(int u, int v, int c, int w)
{
    e[++nume] = Edge(v, c, w, g[u]);
    g[u] = nume;
    e[++nume] = Edge(u, 0, -w, g[v]);
    g[v] = nume;
}

queue<int> que;
bool inQue[maxn + 10];
int dist[maxn + 10];
int prev[maxn + 10], pree[maxn + 10];

bool findPath()
{
    while (!que.empty()) que.pop();
    que.push(src);
    memset(dist, 63, sizeof(dist));
    dist[src] = 0;
    inQue[src] = true;
    while (!que.empty()) {
        int u = que.front();
        que.pop();
        for (int i = g[u]; i; i = e[i].nxt) {
            if (e[i].f > 0 && dist[u] + e[i].w < dist[e[i].v]) {
                dist[e[i].v] = dist[u] + e[i].w;
                prev[e[i].v] = u;
                pree[e[i].v] = i;
                if (!inQue[e[i].v]) {
                    inQue[e[i].v] = true;
                    que.push(e[i].v);
```

```
                    }
                }
            }
            inQue[u] = false;
        }
        if (dist[sink] < inf) return true; else return false;
}

int augment()
{
        int u = sink;
        int delta = inf;
        while (u != src) {
            if (e[pree[u]].f < delta) delta = e[pree[u]].f;
            u = prev[u];
        }
        u = sink;
        while (u != src) {
            e[pree[u]].f -= delta;
            e[pree[u] ^ 1].f += delta;
            u = prev[u];
        }
        return dist[sink] * delta;
}

int mincostflow()
{
        int cur = 0, ans = 0;
        while (findPath()) {
            cur += augment();
            if (cur < ans) ans = cur;
        }
        return ans;
}
```

【注释】

addedge(u, v, c, w)是加边操作，表示加一条u到v容量为c费用为w的边，请务必在开始加边之前把$nume$赋值成1，并把g数组赋值为0。

在SPFA中的memset的无穷大和const的inf应该根据题目要求更改。

【使用范例】

参见程序 POJ2195_2.CPP。

2.6 其　　他

2.6.1 完美消除序列

【任务】

求图的完美消除序列。

【说明】

求图的完美消除序列的最大势（MCS）算法。倒序给点标号，标号为 i 的点出现在完美消除序列的第i项。对于每个顶点i，维护标号$label[i]$，表示为标号的邻接点数量，每次选择标号最大的点进行标号。容易看出，这个过程可以用堆加速，时间复杂度是$O((N+M)\log N)$。

【接口】

vector <int> construct(int n, vector <int> adj[N]);

复杂度：$O((N+M)\log N)$

输　入：n　　　点数

　　　　adj　　　邻接表

输　出：完美消除序列

【代码】

```
const int N = 1111;

vector <int> construct(int n, vector <int> adj[N]) {
    static int rank[N], label[N];
    memset(rank, -1, sizeof(rank));
    memset(label, 0, sizeof(label));
    priority_queue <pair <int, int> > heap;
    for (int i = 0; i < n; ++i) {
        heap.push(make_pair(0, i));
    }
    for (int i = n - 1; i >= 0; --i) {
        while (1) {
            int u = heap.top().second;
```

```
            heap.pop();
            if (rank[u] == -1) {
                rank[u] = i;
                for (vector <int> :: iterator iter = adj[u].begin();
                        iter != adj[u].end(); ++iter) {
                    if (rank[*iter] == -1) {
                        label[*iter]++;
                        heap.push(make_pair(label[*iter], *iter));
                    }
                }
                break;
            }
        }
    }
    vector <int> result(n);
    for (int i = 0; i < n; ++i) {
        result[rank[i]] = i;
    }
    return result;
}
```

【使用范例】

参见程序 ZOJ1015.CPP。

2.6.2　弦图判定

【任务】

判断一个图是否是弦图。

【说明】

用 MCS 算法求出一个完美消除序列，判断是否合法即可。判断的时候，在$v_i, v_{i+1}, \cdots, v_n$的导出子图中找到与$v_i$相邻的标号最小的点，设为$v_j$，再检查$v_j$是否与每个$v_i$的邻接点邻接即可。

【接口】

bool is_chordal(int nodeCount, vector <pair <int, int> > edges);

复杂度：$O(M\log N + N)$，N为点数，M为边数

输　入：*nodeCount*　点数

edges　　图中所有的边，其中边用pair<int, int>表示

输　出：是否为弦图

调用外部函数：

完美消除序列：参见 2.6.1 节

【代码】

```
bool check(int n, vector <int> adj[N], vector <int> ord) {
    static bool mark[N];
    static int rank[N];
    for (int i = 0; i < n; ++i) {
        rank[ord[i]] = i;
    }
    memset(mark, 0, sizeof(mark));
    for (int i = 0; i < n; ++i) {
        vector <pair <int, int> > tmp;
        for (vector <int> :: iterator iter = adj[ord[i]].begin();
                iter != adj[ord[i]].end(); ++iter) {
            if (!mark[*iter]) {
                tmp.push_back(make_pair(rank[*iter], *iter));
            }
        }
        sort(tmp.begin(), tmp.end());
        if (tmp.size()) {
            int u = tmp[0].second;
            set <int> tmpAdj;
            for (vector <int> :: iterator iter = adj[u].begin();
                    iter != adj[u].end(); ++iter) {
                tmpAdj.insert(*iter);
            }
            for (int i = 1; i < (int)tmp.size(); ++i) {
                if (!tmpAdj.count(tmp[i].second)) {
                    return false;
                }
            }
        }
        mark[ord[i]] = true;
    }
    return true;
}
```

```
bool is_chordal(int nodeCount,
        vector <pair <int, int> > edges) {
    int n = nodeCount;
    vector <int> adj[N];
    for (int i = 0; i < n; ++i) {
        adj[i].clear();
    }
    for (vector <pair <int, int> > :: iterator iter = edges.begin();
            iter != edges.end(); ++iter) {
        adj[iter->first].push_back(iter->second);
        adj[iter->second].push_back(iter->first);
    }
    return check(n, adj, construct(n, adj));
}
```

【使用范例】

参见程序 ZOJ1015.CPP。

2.6.3　最大团搜索算法

【任务】

给定一个图，求出一个最大团。

【说明】

令$Si = \{v_i, v_{i+1}, \cdots, v_n\}$，用$mc[i]$表示$\mathrm{MC}(Si)$。倒着算$mc[i]$，那么显然$\mathrm{MC}(V) = mc[1]$。此外有$mc[i] = mc[i+1]$ or $mc[i] = mc[i+1] + 1$，且后一种情况发生的唯一可能是在Si中找到一个包含v_i的团，所以只要搜是不是在Si中存在一个包含v_i且比当前最大团还大的团。

两个剪枝：

（1）$current_size + remain_vertex > ans$

（2）$current_size + mc[i] > ans$

【接口】

void max_cluster( );

输入：g　　全局变量，$g[i][j]$表示i和j之间是否有边相连

输出：ans　　全局变量，保存答案

【代码】

```
void dfs(int size){
    int i, j, k;
    if (len[size]==0) {
        if (size>ans) {
            ans=size;
            found=true;
        }
        return;
    }
    for (k=0; k<len[size] && !found; ++k) {
        if (size+len[size]-k<=ans)
            break;
        i=list[size][k];
        if (size+mc[i]<=ans)
            break;
        for (j=k+1, len[size+1]=0; j<len[size]; ++j)
            if (g[i][list[size][j]])
                list[size+1][len[size+1]++]=list[size][j];
        dfs(size+1);
    }
}

void max_cluster(){
    int i, j;
    mc[n]=ans=1;
    for (i=n-1; i; --i) {
        found=false;
        len[1]=0;
        for (j=i+1; j<=n; ++j)
            if (g[i][j])
                list[1][len[1]++]=j;
        dfs(1);
        mc[i]=ans;
    }
}
```

【注释】

同样的程序可解决求最大独立集问题。求补图的最大团，即是原图的最大独立集。

【使用范例】

参见程序 ZOJ1492.CPP。

2.6.4　极大团的计数

【任务】

极大团的概念：不存在另外一个团包含该团。求图中极大团的个数。

【说明】

算法：搜索+分支定界

搜索方式和一般的搜团的方法一样，回溯的时候如果一个点已经扩展过了，那么以后都不要扩展了（从$Candidate$集合中删去）。但是有一个问题，比如搜到一个团$C_1 = \{v_1, v_2, v_3, v_4\}$，那么假设回溯到第一层，将$v_1$从$Candidate$中删去，此时在搜索就会扩展出团$C_2 = \{v_2, v_3, v_4\}$，而$C_2$不是一个极大团。所以要保持一个$Not$集合，每次$Candidate$集合中一个被踢出来，就要加入$Not$。找到极大团，当且仅当$Candidate$和$Not$都为空。

剪枝：

（1）若当前Not集合中存在一个点，它与$Candidate$中所有点都相连，即在未来的搜索中这个点永远不可能离开Not，那么这就不可能是个极大团，可以剪掉这个分支。具体实现方法可以是，每次从$Candidate$往Not里面添加点时，只检查该点与$Candidate$其他点的连接情况（优点是线性时间内完成，但是有可能有漏掉的）。

（2）这个剪枝是为了最大化1号剪枝的效果。目前我们的算法只是从$Candidate$中随便挑一个点，但是如果我们有选择的挑选，就可以使得这个剪枝效果更好。假设每个在Not里面的顶点都有个cnt值，表示在$Candidate$集合中有多少个点不和它相连。一旦某个cnt为0，那么就达到了1号剪枝的效果。所以2号剪枝的方法就是，选择一个$Candidate$顶点，它和Not中cnt值最小的那个不相连。此外在更新Not和$Candidate$的时候，记住检查当前添加入Not的点是否cnt值更加的小。

【接口】

void cluster_counting( );

输入：g　　全局变量，$g[i][j]$表示i和j之间是否有边相连

输出：ans　　全局变量，保存答案

【代码】

```
void dfs(int size){
    int i, j, k, t, cnt, best = 0;
```

```
    if (ne[size]==ce[size]){
        if (ce[size]==0) ++ans;
        return;
    }
    for (t=0, i=1; i<=ne[size]; ++i) {
        for (cnt=0, j=ne[size]+1; j<=ce[size]; ++j)
            if (!g[list[size][i]][list[size][j]]) ++cnt;
        if (t==0 || cnt<best) t=i, best=cnt;
    }
    if (t && best<=0)
        return;
    for (k=ne[size]+1; k<=ce[size]; ++k) {
        if (t>0){
            for (i=k; i<=ce[size]; ++i)
                if (!g[list[size][t]][list[size][i]])
                    break;
            swap(list[size][k], list[size][i]);
        }
        i=list[size][k];
        ne[size+1]=ce[size+1]=0;
        for (j=1; j<k; ++j)
            if (g[i][list[size][j]])
                list[size+1][++ne[size+1]]=list[size][j];
        for (ce[size+1]=ne[size+1], j=k+1; j<=ce[size]; ++j)
            if (g[i][list[size][j]])
                list[size+1][++ce[size+1]]=list[size][j];
        dfs(size+1);
        ++ne[size];
        --best;
        for (j=k+1, cnt=0; j<=ce[size]; ++j)
            if (!g[i][list[size][j]])
                ++cnt;
        if (t==0 || cnt<best) t=k, best=cnt;
        if (t && best<=0) break;
    }
}

void cluster_counting(){
    int i;
```

```
    ne[0]=0;
    ce[0]=0;
    for (i=1; i<=n; ++i)
        list[0][++ce[0]]=i;
    ans=0;
    dfs(0);
}
```

【使用范例】

参见程序 POJ2989.CPP。

2.6.5　图的同构

【任务】

给出两个有向图（无向图可以转化为有向图做），判断两个图是否是同构的。

【说明】

采用hash的办法，对于两个图中的每个点分别求出一个hash值，若两个图的hash值相同则说明图是同构的。

Hash函数需要体现点与周围点的连边关系，程序中使用的hash函数如下：

$$F_t(i)=\left(F_{t-1}(i)\times A+\sum_{i\to j}F_{t-1}(j)\times B+\sum_{j\to i}F_{t-1}(j)\times C+D\times(i==a)\right)\bmod P$$

枚举点a，迭代K次后求得的$F_K(a)$就是a点所对应的hash值。

其中K、A、B、C、D、P为hash参数，可自选。

【接口】

void graph_hash( );

复杂度：O(nmK)

输　入：n,m　　全局变量，图的点数、边数

　　　　a　　　全局变量，$a[i][]$图中第i条边的两个顶点

输　出：co　　全局变量，每个点的hash值

【代码】

```
int a[100000][2];
int co[1000];
int f[1000],tf[1000];
int n,m;
```

```

void graph_hash(){
    int q,w,e;
    for (int i=0;i<n;i++){
        for (q=0;q<n;q++) f[q]=1;
        for (int z=0;z<K;z++){
            memcpy(tf,f,sizeof(f));
            for (q=0;q<n;q++) f[q]*=A;
            for (q=0;q<m;q++){
                f[a[q][0]]+=tf[a[q][1]]*B;
                f[a[q][1]]+=tf[a[q][0]]*C;
            }
            f[i]+=D;
            for (q=0;q<n;q++) f[q]%=P;
        }
        co[i]=f[i];
    }
    sort(co,co+n);
}
```

【使用范例】

参见程序 USTC1119.CPP。

2.6.6 树的同构

【任务】

给出两棵有根树，判断是否同构。

【说明】

对树的括号序列作Rabin-Karp即可。因为子树顺序没有影响，所以应该先排序。

【接口】

unsigned long long getHash(int root, vector <pait <int, int> > &edges);

复杂度：$O(n\log n)$

输　入：*root*　　树根的标号

　　　　edges　　树的边集

输　出：树的散列值

【代码】

```
typedef unsigned long long ULL;

const int maxNode = 111111;
const ULL MAGIC = 321;

ULL powMod(ULL a, int n) {
    ULL ret = 1ULL;
    while (n) {
        if (n & 1) {
            ret *= a;
        }
        a *= a;
        n >>= 1;
    }
    return ret;
}

struct Hash {
    int length;
    ULL value;

    Hash(): length(0), value(0) {}
    Hash(char c): length(1), value(c) {}
    Hash(int l, ULL v): length(l), value(v) {}
};

bool operator <(const Hash &a, const Hash &b) {
    return a.value < b.value;
}

Hash operator +(const Hash &a, const Hash &b) {
    return Hash(a.length + b.length,
            a.value * powMod(MAGIC, b.length) + b.value);
}

void operator +=(Hash &a, const Hash &b) {
    a = a + b;
```

```
}

int edgeCnt, firstEdge[maxNode], to[maxNode << 1],
    nextEdge[maxNode << 1];

void addEdge(int u, int v) {
    to[edgeCnt] = v;
    nextEdge[edgeCnt] = firstEdge[u];
    firstEdge[u] = edgeCnt++;
}

vector <Hash> childs[maxNode];

Hash dfs(int pre, int cur) {
    Hash ret;
    childs[cur].clear();
    for (int iter=firstEdge[cur]; iter != -1; iter = nextEdge[iter]) {
        if (to[iter] != pre) {
            childs[cur].push_back(dfs(cur, to[iter]));
        }
    }
    sort(childs[cur].begin(), childs[cur].end());
    for (vector <Hash> :: iterator iter = childs[cur].begin();
            iter != childs[cur].end(); ++iter) {
        ret += *iter;
    }
    ret = '(' + ret + ')';
    return ret;
}

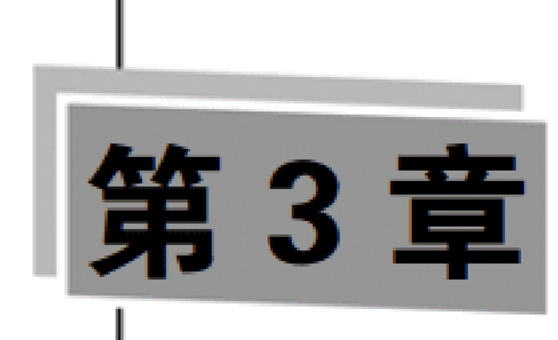
ULL getHash(int root, vector <pair <int, int> >& edges) {
    edgeCnt = 0;
    memset(firstEdge, -1, sizeof(firstEdge));
    for (vector <pair <int, int> > :: iterator iter = edges.begin();
            iter != edges.end(); ++iter) {
        addEdge(iter->first, iter->second);
        addEdge(iter->second, iter->first);
    }
    return dfs(-1, root).value;
}
```

【注释】

如果两棵无根树同构，那么把它们的重心选为根形成的有根树也一定同构。所以只需选出重心，然后套用有根树的方法即可。

【使用范例】

参见程序 POJ1635.CPP。



计 算 几 何

3.1 多 边 形

3.1.1 计算几何误差修正

【任务】

给定一个double类型的数，判断它的符号。

【说明】

因为计算几何中经常涉及精度问题，需要对一个很小的数判断正负，所以需要引入一个极小量eps。

【接口】

int cmp(double x);

输入： x 判断符号的数

输出： x的符号，-1表示x为负数，1表示x为正数，0表示x为0。

【代码】

```
const double eps = 1e-8;
int cmp(double x) {
    if (fabs(x) < eps) return 0;
    if (x > 0) return 1;
    return -1;
}
```

【注释】

在本章中，常用基本类型（如 point）和常用基本函数（如 cmp）就不额外标出外部调用了。

【使用范例】

参见程序 POJ2653.CPP。

3.1.2　计算几何点类

【任务】

设计一个二维点类，可以进行一些向量运算。

【接口】

结构体：*point*

成员变量：

double x, y　　点的坐标

重载运算符：+, −, ×, /, ==

成员函数：

input()　　输入一个点

norm()　　计算向量的模长

相关函数：

double sqr(double x)　　计算一个数的平方

double det(const point &a, const point &b)　　计算两个向量的叉积

double dot(const point &a, const point &b)　　计算两个向量的点积

double dist(const point &a, const point &b)　　计算两个点的距离

point rotate_point(const point &p, double A)　　$\overrightarrow{op}$绕原点逆时针旋转A（弧度）

【代码】

```
const double pi = acos(-1.0);
inline double sqr(double x) {
    return x * x;
}
struct point {
    double x, y;
    point() {}
    point(double a, double b): x(a), y(b) {}
    void input() {
        scanf("%lf%lf", &x, &y);
    }
    friend point operator + (const point &a, const point &b) {
```

```
        return point(a.x + b.x, a.y + b.y);
    }
    friend point operator - (const point &a, const point &b) {
        return point(a.x - b.x, a.y - b.y);
    }
    friend bool operator == (const point &a, const point &b) {
        return cmp(a.x - b.x) == 0 && cmp(a.y - b.y) == 0;
    }
    friend point operator * (const point &a, const double &b) {
        return point(a.x * b, a.y * b);
    }
    friend point operator * (const double &a, const point &b) {
        return point(a * b.x, a * b.y);
    }
    friend point operator / (const point &a,const double &b) {
        return point(a.x / b, a.y / b);
    }
    double norm() {
        return sqrt(sqr(x) + sqr(y));
    }
};
double det(const point &a, const point &b) {
    return a.x * b.y - a.y * b.x;
}
double dot(const point &a, const point &b) {
    return a.x * b.x + a.y * b.y;
}
double dist(const point &a, const point &b) {
    return (a - b).norm();
}
point rotate_point(const point &p, double A) {
    double tx = p.x, ty = p.y;
    return point(tx * cos(A) - ty * sin(A), tx * sin(A) + ty * cos(A));
}
```

【使用范例】

参见程序 POJ2653.CPP。

3.1.3 计算几何线段类

【任务】

实现一个线段类，可以完成线段的一些计算几何运算。

【说明】

为了避免精度问题，且实现起来方便，线段用一个有向线段表示。线段类的运算也都使用向量运算。

在存储时，就存下线段上的两点，用$a \to b$来表示有向线段。同样也可以用这种方式来表示直线。

【接口】

结构体：*line*

成员变量：

point a, b　线段的两个端点

相关函数：

line point_make_line(const point a,const point b);

用两个点a, b生成的一个线段或者直线

double dis_point_segment(const point p,const point s,const point t);

求p点到线段st的距离

void PointProjLine(const point p, const point s, const point t, point &cp);

求p点到线段st的垂足，保存在cp中。

bool PointOnSegment (point p, point s, point t);

判断p点是否在线段st上（包括端点）

bool parallel(line a,line b);

判断a和b是否平行。

bool line_make_point(line a,line b,point &res);

判断a和b是否相交，如果相交则返回true且交点保存在res中

line move_d(line a,const double &len);

将直线a沿法向量方向平移距离len得到的直线

【代码】

```
struct line {
    point a,b;
    line() {}
```

```
    line(point x,point y): a(x),b(y) {}
};
line point_make_line(const point a,const point b) {
    return line(a,b);
}
double dis_point_segment(const point p,const point s,const point t) {
    if (cmp(dot(p-s,t-s))<0) return (p-s).norm();
    if (cmp(dot(p-t,s-t))<0) return (p-t).norm();
    return fabs(det(s-p,t-p)/dist(s,t));
}
void PointProjLine(const point p,const point s, const point t, point &cp){
    double r=dot((t-s),(p-s))/dot(t-s,t-s);
    cp=s+r*(t-s);
}
bool PointOnSegment (point p, point s, point t) {
   return cmp(det(p-s,t-s))==0 && cmp(dot(p-s,p-t))<=0;
}
bool parallel(line a,line b) {
    return !cmp(det(a.a-a.b,b.a-b.b));
}
bool line_make_point(line a,line b,point &res) {
    if (parallel(a,b)) return false;
    double s1=det(a.a-b.a,b.b-b.a);
    double s2=det(a.b-b.a,b.b-b.a);
    res=(s1*a.b-s2*a.a)/(s1-s2);
    return true;
}
line move_d(line a,const double &len) {
    point d=a.b-a.a;
    d=d/d.norm();
    d=rotate_point(d,pi/2);
    return line(a.a+d*len,a.b+d*len);
}
```

【使用范例】

参见程序 POJ2653.CPP，POJ1584.CPP。

3.1.4 多边形类

【任务】

实现一个多边形类，完成计算多边形的面积、重心等基本操作。

【说明】

判断点在多边形内：从该点做一条水平向右的射线，统计射线与多边形相交的情况，若相交次数为偶数，则说明该点在形外，否则在形内。为了便于交点在顶点或射线与某些边重合时的判断，可以将每条边看成左开右闭的线段，即若交点为左端点则不计算。

【接口】

结构体：*polygon*

成员变量：

int n	多边形点数
point a[ ]	多边形顶点坐标（按顺时针顺序）

成员函数：

double perimeter( )	计算多边形周长
double area( )	计算多边形面积
int Point_In(point t);	判断点是否在多边形内部

复杂度：$O(N)$

输　入：t　需要判断的点t

输　出：0表示t点在多边形外

1表示t点在多边形内

2表示在t点在多边形的边界上

【代码】

```
const int maxn = 100;
struct polygon {
    int n;
    point a[maxn];
    polygon(){}
    double perimeter() {
        double sum=0;
        a[n]=a[0];
        for (int i=0;i<n;i++) sum+=(a[i+1]-a[i]).norm();
        return sum;
```

```
    }
    double area() {
        double sum=0;
        a[n]=a[0];
        for (int i=0;i<n;i++) sum+=det(a[i+1],a[i]);
        return sum/2.;
    }
    int Point_In(point t) {
        int num=0,i,d1,d2,k;
        a[n]=a[0];
        for (i=0;i<n;i++){
            if (PointOnSegment(t,a[i],a[i+1])) return 2;
            k=cmp(det(a[i+1]-a[i],t-a[i]));
            d1=cmp(a[i].y-t.y);
            d2=cmp(a[i+1].y-t.y);
            if (k>0 && d1<=0 && d2>0) num++;
            if (k<0 && d2<=0 && d1>0) num--;
        }
        return num!=0;
    }
};
```

【使用范例】

参见程序 POJ1584.CPP，ZOJ1081.CPP。

3.1.5 多边形的重心

【任务】

多边形类中的成员函数，求多边形的重心。

【说明】

将多边形分割为三角形的并，对每个三角形求重心（三角形重心即为三点坐标的平均值），然后以三角形的有向面积为权值求加权平均即可。

【接口】

point polygon::MassCenter();

复杂度：$O(N)$

输　出：多边形的重心坐标

【代码】

```
point polygon::MassCenter() {
    point ans=point(0,0);
    if (cmp(area())==0) return ans;
    a[n]=a[0];
    for (int i=0;i<n;i++) ans=ans+(a[i]+a[i+1])*det(a[i+1],a[i]);
    return ans/area()/6.;
}
```

【注释】

当多边形面积为0时重心没有定义，需要特别处理。

【使用范例】

参见程序 POJ1385.CPP。

3.1.6 多边形内格点数

【任务】

给出多边形的顶点（整点），求多边形内以及多边形边界上格点的个数。

【说明】

Pick公式：

给定顶点坐标均是整点的简单多边形，有：

$$面积=内部格点数目+边上格点数目/2-1$$

边界上的格点数：

把每条边当做左开右闭的区间以避免重复，一条左开右闭的线段$(x1, y1) \rightarrow (x2, y2)$上的格点数为：$\gcd(x2 - x1, y2 - y1)$。

【接口】

polygon::Border_Int_Point_Num( );

输入：a 全局变量，表示多边形顶点坐标

输出：多边形边界上的格点个数

polygon::Inside_Int_Point_Num( );

输入：a 全局变量，表示多边形顶点坐标

输出：多边形内的格点个数

【代码】

```
int polygon::Border_Int_Point_Num() {
    int num=0;
    a[n]=a[0];
    for (int i=0;i<n;i++)
        num+=gcd(abs(int(a[i+1].x-a[i].x)),abs(int(a[i+1].y
        -a[i].y)));
    return num;
}
int polygon::Inside_Int_Point_Num(){
    return int(area())+1-Border_Int_Point_Num()/2;
}
```

【使用范例】

参见程序 POJ1265.CPP。

3.1.7 凸多边形类

【任务】

实现一个凸多边形类，可以求出凸包、判断点是否在凸包内。

【说明】

为了避免精度问题，求凸包采用的是水平序的求法。

判断点是否在凸包内实现了一个复杂度$O(n)$的和一个复杂度$O(\log n)$的。

【接口】

polygon_convex convex_hull(vector<point> a);

复杂度：$O(n\log n)$

输　入：a　所有的点

输　出：用a中的点求出的凸包（逆时针顺序）

bool containOn(const polygon_convex &a,const point &b);

复杂度：$O(n)$

输　入：&a　一个凸包

　　　　&b　一个点

输　出：点b是否在凸包a中，true表示点在凸包内部或者在边界上

int containOlogn(const polygon_convex &a,const point &b);

复杂度：$O(\log n)$

输　入：$\&a$　一个凸包

　　　　$\&b$　一个点

输　出：点b是否在凸包a中，true表示点在凸包内部或者在边界上

【代码】

```
struct polygon_convex {
    vector <point> P;
    polygon_convex(int Size=0) {
        P.resize(Size);
    }
};

bool comp_less(const point &a,const point &b) {
    return cmp(a.x-b.x)<0 || cmp(a.x-b.x)==0 && cmp(a.y-b.y)<0;
}
polygon_convex convex_hull(vector<point> a) {
    polygon_convex res(2*a.size()+5);
    sort(a.begin(),a.end(),comp_less);
    a.erase(unique(a.begin(),a.end()),a.end());
    int m=0;
    for (int i=0;i<a.size();++i){
        while (m>1 && cmp(det(res.P[m-1]-res.P[m-2],a[i]-res.P[m-2]))
        <=0) --m;
        res.P[m++]=a[i];
    }
    int k=m;
    for (int i=int(a.size())-2;i>=0;--i) {
        while (m>k && cmp(det(res.P[m-1]-res.P[m-2],a[i]-res.P[m-2]))
        <=0) --m;
        res.P[m++]=a[i];
    }
    res.P.resize(m);
    if (a.size()>1) res.P.resize(m-1);
    return res;
}

bool containOn(const polygon_convex &a,const point &b) {
    int n=a.P.size();
```

```
    #define next(i) ((i+1)%n)
    int sign=0;
    for (int i=0;i<n;++i) {
        int x=cmp(det(a.P[i]-b,a.P[next(i)]-b));
        if (x) {
            if (sign) {
                if (sign!=x) return false;
            } else sign=x;
        }
    }
    return true;
}

int containOlogn(const polygon_convex &a,const point &b) {
    int n=a.P.size();
    //找一个凸包内部的点 g
    point g=(a.P[0]+a.P[n/3]+a.P[2*n/3])/3.0;
    int l=0,r=n;
    //二分凸包 g-a.P[a]-a.P[b]
    while (l+1<r) {
        int mid=(l+r)/2;
        if (cmp(det(a.P[l]-g,a.P[mid]-g))>0) {
            if (cmp(det(a.P[l]-g,b-g))>=0 && cmp(det(a.P[mid]-g,b-g))
            <0) r=mid;
            else l=mid;
        }else {
            if (cmp(det(a.P[l]-g,b-g))<0 && cmp(det(a.P[mid]-g,b-g))
            >=0) l=mid;
            else r=mid;
        }
    }
    r%=n;
    int z=cmp(det(a.P[r]-b,a.P[l]-b))-1;
    if (z==-2) return 1;
    return z;
}
```

【使用范例】

参见程序 POJ1228.CPP，POJ1264.CPP，POJ2187.CPP。

3.1.8 凸多边形的直径

【任务】

传入一个凸包，求出欧几里得距离最远的两点。

【说明】

使用旋转卡壳算法。

【接口】

double convex_diameter(polygon_convex &a,int &First,int &Second);

复杂度：$O(n)$

输　入：$\&a$　　凸包

输　出：$\&First, \&Second$　　最远两个点的对应标号

凸包上最远欧几里得距离

【代码】

```
double convex_diameter(polygon_convex &a,int &First,int &Second) {
    vector<point> &p=a.P;
    int n=p.size();
    double maxd=0.0;
    if (n==1) {
        First=Second = 0;
        return maxd;
    }
    #define next(i) ((i+1)%n)
    for (int i=0,j=1;i<n;++i) {
        while (cmp(det(p[next(i)]-p[i],p[j]-p[i])-det(p[next(i)]
        -p[i],p[next(j)]-p[i]))<0)
        {
            j=next(j);
        }
        double d=dist(p[i],p[j]);
        if (d>maxd){
            maxd=d;
            First=i,Second=j;
        }
        d=dist(p[next(i)],p[next(j)]);
        if (d>maxd){
```

```
                maxd=d;
                First=i,Second=j;
            }
        }
        return maxd;
    }
```

【使用范例】

参见程序 POJ2187.CPP。

3.1.9 半平面切割多边形

【任务】

给定一个半平面和一个多边形，求它们的交。

【说明】

做法是用给定的半平面去切割凸多边形。

【接口】

polygon_convex cut(polygon_convex &a,halfPlane &L)

复杂度：O(N)

输　入：&a　一个凸多边形

　　　　&L　一个半平面

输　出：一个凸多边形

【代码】

```
struct halfPlane
{
    //ax+by+c<=0
    double a,b,c;

    halfPlane(point p, point q)
    {
        a = p.y - q.y;
        b = q.x - p.x;
        c = det(p, q);
    }
    halfPlane(double aa,double bb,double cc)
```

```
    {
        a=aa;
        b=bb;
        c=cc;
    }
};

//计算点 a 带入到直线方程中的函数值
double calc(halfPlane &L,point &a)
{
    return a.x*L.a+a.y*L.b+L.c;
}

//求点 a 和 b 连线与半平面 L 的交点
point Intersect(point &a,point &b,halfPlane &L)
{
    point res;
    double t1=calc(L,a),t2=calc(L,b);
    res.x=(t2*a.x-t1*b.x)/(t2-t1);
    res.y=(t2*a.y-t1*b.y)/(t2-t1);
    return res;
}

//将一个凸多边形和一个半平面交
polygon_convex cut(polygon_convex &a,halfPlane &L)
{
    int n=a.P.size();
    polygon_convex res;
    for (int i=0;i<n;++i)
    {
        if (calc(L,a.P[i])<-eps) res.P.push_back(a.P[i]);
        else
        {
            int j;
            j=i-1;
            if (j<0) j=n-1;
            if (calc(L,a.P[j])<-eps)
                res.P.push_back(Intersect(a.P[j],a.P[i],L));
            j=i+1;
```

```
                if (j==n) j=0;
                if (calc(L,a.P[j])<-eps)
                    res.P.push_back(Intersect(a.P[i],a.P[j],L));
            }
        }
        return res;
    }
```

【使用范例】

参见程序 POJ1279.CPP。

3.1.10 半平面交

【任务】

给定n个半平面，求出它们的交。

【说明】

我们用一个向量$(x_1,y_1) \to (x_2,y_2)$的左侧来描述一个半平面。首先将半平面按极角排序，极角相同的则只保留最左侧的一个。然后用一个双端队列维护这些半平面：按顺序插入，在插入半平面p_i之前判断双端队列尾部的两个半平面的交是否在半平面p_i内，如果不是则删除最后一个半平面；判断双端队列顶部的两个半平面的交是否在半平面p_i内，如果不是则删除第一个半平面。插入完毕之后再处理一下双端队列两端多余的半平面，最后求出尾端和顶端的两个半平面的交点即可。

【接口】

vector<Point> halfplaneIntersection(vector<Halfplane> v);

复杂度：O($n\log n$)

输　入：v　一组半平面

输　出：一个凸多边形，表示这些半平面的交

【代码】

```
typedef complex<double> Point;
typedef pair<Point, Point> Halfplane;
const double EPS = 1e-10;
const double INF = 10000;

inline int sgn(double n) { return fabs(n) < EPS ? 0 : (n < 0 ? -1 : 1); }
```

```
inline double cross(Point a, Point b) { return (conj(a) * b).imag(); }
inline double dot(Point a, Point b) { return (conj(a) * b).real(); }
inline double satisfy(Point a, Halfplane p) {
    return sgn(cross(a - p.first, p.second - p.first)) <= 0;
}

Point crosspoint(const Halfplane &a, const Halfplane &b) {
    double k = cross(b.first - b.second, a.first - b.second);
    k = k / (k - cross(b.first - b.second, a.second - b.second));
    return a.first + (a.second - a.first) * k;
}

bool cmp(const Halfplane &a, const Halfplane &b) {
    int res = sgn(arg(a.second - a.first) - arg(b.second - b.first));
    return res == 0 ? satisfy(a.first, b) : res < 0;
}

vector<Point> halfplaneIntersection(vector<Halfplane> v) {
    sort(v.begin(), v.end(), cmp);
    deque<Halfplane> q;
    deque<Point> ans;
    q.push_back(v[0]);
    for (int i = 1; i <int(v.size()); ++i) {
        if (sgn(arg(v[i].second - v[i].first) - arg(v[i - 1].second -
        v[i - 1].first)) == 0) {
            continue;
        }
        while (ans.size() > 0 && !satisfy(ans.back(), v[i])) {
            ans.pop_back();
            q.pop_back();
        }
        while (ans.size() > 0 && !satisfy(ans.front(), v[i])) {
            ans.pop_front();
            q.pop_front();
        }
        ans.push_back(crosspoint(q.back(), v[i]));
        q.push_back(v[i]);
    }
    while (ans.size() > 0 && !satisfy(ans.back(), q.front())) {
```

```
            ans.pop_back();
            q.pop_back();
        }
        while (ans.size() > 0 && !satisfy(ans.front(), q.back())) {
            ans.pop_front();
            q.pop_front();
        }
        ans.push_back(crosspoint(q.back(), q.front()));
        return vector<Point>(ans.begin(), ans.end());
    }
```

【注释】

代码中采用了complex<double>来表示一个点Point，pair<Point,Point>来表示一个半平面Halfplane。

【使用范例】

参见程序 POJ2451.CPP。

3.1.11 凸多边形交

【任务】

给定两个凸多边形，求它们的交。

【说明】

直接套用半平面交算法，将每个凸多边形分解为一组半平面，求它们的交。

【接口】

Convex convexIntersection(Convex v1, Convex v2);

复杂度：取决于所调用的半平面交算法的复杂度

输　入：$v1, v2$　两个凸多边形

输　出：一个凸多边形，表示它们的交

【代码】

```
typedef complex<double> Point;
typedef pair<Point, Point> Halfplane;
typedef vector<Point> Convex;

Convex convexIntersection(Convex v1, Convex v2) {
```

```
    vector<Halfplane> h;
    for (int i = 0; i <int(v1.size()); ++i)
        h.push_back(Halfplane(v1[i], v1[(i + 1) % v1.size()]));
    for (int i = 0; i <int(v2.size()); ++i)
        h.push_back(Halfplane(v2[i], v2[(i + 1) % v2.size()]));
    return halfplaneIntersection(h);
}
```

【注释】

一个线性做凸多边形交的想法是：由于多边形的边的斜率是有序的，于是可以做一次线性归并将它们排序。之后套用 3.1.11 节的算法。3.1.11 节的算法除了排序的部分也是线性的。

【使用范例】

参见程序 ECNU1624.CPP。

3.1.12　多边形的核

【任务】

求一个多边形的核。

【说明】

直接套用半平面交来计算。

inf 根据坐标范围而定，足够大即可。

【接口】

polygon_convex core(polygon &a);

复杂度：取决于半平面交算法的时间复杂度

输　入：&a　一个多边形

输　出：一个凸多边形，表示a的核

【代码】

```
polygon_convex core(polygon &a)
{
    polygon_convex res;
    res.P.push_back(point(-inf,-inf));
    res.P.push_back(point(inf,-inf));
    res.P.push_back(point(inf,inf));
```

```
        res.P.push_back(point(-inf,inf));

        int n=a.n;
        for (int i=0;i<n;++i)
        {
            halfPlane L(a.a[i],a.a[(i+1)%n]);
            res=cut(res,L);
        }
        return res;
    }
```

【使用范例】

参见程序 POJ1279.CPP。

3.1.13 凸多边形与直线集交

【任务】

给定m条直线和一个n个点的凸包。求这些直线是否和凸包有交点。

【说明】

算法大致是这样的：对于每条直线，可以看成两个方向相反的向量，把凸包上的边也看成向量（不妨假设凸包是逆时针的）。对于每个向量，我们都可以用二分法找到它左侧的第一个向量。于是，我们可以在凸包的边中，分别找到在直线两侧的点。凸包就可以分为两半，每一半都有一个交点。这个交点也可以通过二分法找到。

【接口】

void GetHull( );

复杂度：O($n$log$n$)

输　出：预处理出n个点的凸包。

inline bool solve(point P,point Q);

复杂度：O(log$n$)

输　入：P,Q　　直线的两个点

输　出：直线是否与凸包有交点

【代码】

```
inline bool operator <(const point &a,const point &b){
    return a.y+eps<b.y || fabs(a.y-b.y)<eps && a.x+eps<b.x;
```

```
}
inline double getA(const point &a)      //凸包顺序下对应的角度也要递增
{
    double res=atan2(a.y,a.x);
    if (res<0) res+=2*pi;
    return res;
}

point p[maxn],hull[maxn];
int n;
double w[maxn],sum[maxn];

inline void GetHull()                   //预处理一个逆时针的凸包
{
    sort(p+1,p+n+1);
    int N=0;
    hull[++N]=p[1];
    for (int i=2;i<=n;++i)
    {
        while (N>1 && det(hull[N]-hull[N-1],p[i]-hull[N-1])<=0) --N;
        hull[++N]=p[i];
    }
    int bak=N;
    for (int i=n-1;i>=1;--i)
    {
        while (N>bak && det(hull[N]-hull[N-1],p[i]-hull[N-1])<=0) --N;
        hull[++N]=p[i];
    }
    n=N-1;
    for (int i=1;i<=n;++i)
        p[i+n]=p[i]=hull[i];
    p[n+n+1]=p[1];

    for (int i=1;i<=n;++i)
        w[i+n]=w[i]=getA(p[i+1]-p[i]);

    sum[0]=0;
    for (int i=1;i<=2*n;++i)
        sum[i]=sum[i-1]+det(p[i],p[i+1]);       //预处理有向面积前缀和
```

```
}

inline int Find(double x)                        //找第一个角度>x 的边
{
    if (x<=w[1] || x>=w[n]) return 1;
    return (upper_bound(w+1,w+n+1,x)-(w+1))+1;
}

point P,Q;

inline int getInter(int l,int r)//找到第一个和 p[l]不在 P→Q 向量同侧的点
{
    int sign;
    if (det(Q-P,p[l]-P)<0) sign=-1;
    else sign=1;
    while (l+1<r)
    {
        int mid=(l+r)/2;
        if (det(Q-P,p[mid]-P)*sign>0) l=mid;
        else r=mid;
    }
    return r;
}

inline point Intersect(const point &a,const point &b,const point &c,
const point &d)
//两直线求交点
{
    double s1=det(c-a,b-a);
    double s2=det(d-a,b-a);
    return (c*s2-d*s1)/(s2-s1);
}

inline bool solve(point P,point Q)
{
    int i=Find(getA(Q-P));
    int j=Find(getA(P-Q));
    //两侧各找一点
    if (det(Q-P,p[i]-P)*det(Q-P,p[j]-P)>=0)          //无交点
```

```
            return false;
        else
            return true;
}
```

【使用范例】

参见程序 POJ1912.CPP。

3.2　圆

3.2.1　圆与线求交

【任务】

求圆与线段（直线）的交点。

【说明】

将线段$\overrightarrow{AB}$写成参数方程 $\vec{P} = \vec{A} + t \cdot (\vec{B} - \vec{A})$，带入圆的方程，得到一个一元二次方程。解出$t$就可以求得线段所在的直线与圆的交点。如果$0 \leqslant t \leqslant 1$则说明点在线段上。下面的代码求出来的两个交点（如果有的话）P_0, P_1构成的向量 $\overrightarrow{P_0P_1}$与 $\overrightarrow{AB}$同向。

【接口】

void circle_cross_line(point a, point b, point o, double r, point ret[], int &num);

输入：a, b	表示线段（直线）$\overrightarrow{ab}$
o	圆心
r	圆的半径
ret	函数的计算出来的交点
$\&num$	引用，函数计算出来的交点个数

【代码】

```
void circle_cross_line(point a, point b, point o, double r, point ret[],
int &num) {
    double x0 = o.x, y0 = o.y;
    double x1 = a.x, y1 = a.y;
    double x2 = b.x, y2 = b.y;
    double dx = x2 - x1, dy = y2 - y1;
    double A = dx*dx + dy*dy;
    double B = 2*dx*(x1 - x0) + 2*dy*(y1 - y0);
    double C = sqr(x1 - x0) + sqr(y1 - y0) - sqr(r);
```

```
    double delta = B*B - 4*A*C;
    num = 0;
    if (dcmp(delta) >= 0) {
        double t1 = (-B - mysqrt(delta)) / (2*A);
        double t2 = (-B + mysqrt(delta)) / (2*A);
        if (dcmp(t1 - 1) <= 0 && dcmp(t1) >= 0) {
            ret[num++] = point(x1 + t1*dx, y1 + t1*dy);
        }
        if (dcmp(t2 - 1) <= 0 && dcmp(t2) >= 0) {
            ret[num++] = point(x1 + t2*dx, y1 + t2*dy);
        }
    }
}
```

【注释】

把代码中判断t的范围的两个if去掉，就可以计算圆与直线的交点。

【使用范例】

参见程序 POJ3675.CPP。

3.2.2 圆与多边形交的面积

【任务】

求圆与简单多边形的交的面积。圆心处于原点。

【说明】

按原点为中心将多边形三角剖分，这样只需要求一个三角形和圆的交的面积。对于三角形OAB，有以下4种情况：

（1）AB都在圆内，此时计算三角形OAB的面积即可；

（2）A在圆内B不在圆内，此时计算一个三角形的面积和一个扇形的面积；

（3）A不在圆内B在圆内，此时计算一个三角形的面积和一个扇形的面积；

（4）AB都不在圆内，若AB与圆无交点，则计算一个扇形的面积，否则要计算两个扇形的面积和一个三角形的面积。

【接口】

double area( );

复杂度：$O(n)$

输　入：n　　全局变量，多边形的点数

　　　　res　全局变量，逆时针存入多边形的所有点

　　　　r　　全局变量，圆的半径

输　出：圆与多边形的交的面积

调用外部函数：

　　圆与线：参见 3.2.1 节。

【代码】

```
int dcmp(double k) {
    return k < -EPS ? -1 : k > EPS ? 1 : 0;
}

double dot(const point &a, const point &b) {
    return a.x * b.x + a.y * b.y;
}

double cross(const point &a, const point &b) {
    return a.x * b.y - a.y * b.x;
}

double abs(const point &o) {
    return sqrt(dot(o, o));
}

point crosspt(const point &a, const point &b, const point &p, const
point &q) {
    double a1 = cross(b - a, p - a);
    double a2 = cross(b - a, q - a);
    return (p * a2 - q * a1) / (a2 - a1);
}

point res[MAXN];
double r;
int n;

double mysqrt(double n) {
    return sqrt(max(0.0, n));
}
```

```

double sector_area(const point &a, const point &b) {
    double theta = atan2(a.y, a.x) - atan2(b.y, b.x);
    while (theta <= 0) theta += 2*PI;
    while (theta > 2*PI) theta -= 2*PI;
    theta = min(theta, 2*PI - theta);
    return r * r * theta / 2;
}

double calc(const point &a, const point &b) {
    point p[2];
    int num = 0;
    int ina = dcmp(abs(a) - r) < 0;
    int inb = dcmp(abs(b) - r) < 0;
    if (ina) {
        if (inb) {
            return fabs(cross(a, b)) / 2.0;
        } else {
            circle_cross_line(a, b, point(0, 0), r, p, num);
            return sector_area(b, p[0]) + fabs(cross(a, p[0])) / 2.0;
        }
    } else {
        if (inb) {
            circle_cross_line(a, b, point(0, 0), r, p, num);
            return sector_area(p[0], a) + fabs(cross(p[0], b)) / 2.0;
        } else {
            circle_cross_line(a, b, point(0, 0), r, p, num);
            if (num == 2) {
                return sector_area(a, p[0]) + sector_area(p[1], b)
                     + fabs(cross(p[0], p[1])) / 2.0;
            } else {
                return sector_area(a, b);
            }
        }
    }
}

double area() {
    double ret = 0;
```

```
    for (int i = 0; i < n; ++i) {
        int sgn = dcmp(cross(res[i], res[i + 1]));
        if (sgn != 0) {
            ret += sgn * calc(res[i], res[i + 1]);
        }
    }
    return ret;
}
```

【注释】

注意需要保证$res[n] = res[0]$。

【使用范例】

参见程序 POJ3675.CPP。

3.2.3 最小圆覆盖

【任务】

要求一个半径最小的圆覆盖住所有的点。

【说明】

使用随机增量的方法，每次找到一个不在当前圆内的点，将圆调整扩大至该点在圆周上。期望复杂度为$O(n)$。注意在调用min_circle_cover之前应先把a[]中的点打乱顺序。

【接口】

void min_circle_cover(point a[],int n)

复杂度：期望$O(n)$

输　入：a　　需要覆盖的所有的点

n　　点的个数

输　出：$center$　　全局变量，最小覆盖圆的圆心

$radius$　　全局变量，最小覆盖圆的半径

【代码】

```
void circle_center(point p0 , point p1 , point p2 , point &cp){
    double a1=p1.x-p0.x , b1=p1.y-p0.y , c1=(a1*a1+b1*b1) / 2 ;
    double a2=p2.x-p0.x , b2=p2.y-p0.y , c2=(a2*a2+b2*b2) / 2 ;
    double d = a1*b2 - a2*b1 ;
    cp.x = p0.x + (c1*b2 - c2*b1) / d ;
```

```
    cp.y = p0.y + (a1*c2 - a2*c1) / d ;
}

void circle_center(point p0 , point p1 , point &cp ){
     cp.x=(p0.x+p1.x)/2;
     cp.y=(p0.y+p1.y)/2;
}

point center;
double radius;

bool point_in(const point &p) {
    return cmp((p - center).dist() - radius) < 0;
}

void min_circle_cover(point a[],int n){
    radius = 0;
    center = a[0];
     for (int i=1; i<n; i++) if (!point_in(a[i])){
         center = a[i]; radius = 0;
         for (int j=0; j<i; j++) if (!point_in(a[j])){
             circle_center(a[i], a[j], center);
             radius = (a[j] - center).dist();
             for (int k=0;k<j;k++) if (!point_in(a[k])){
                 circle_center(a[i], a[j], a[k], center);
                 radius = (a[k] - center).dist();
             }
         }
     }
}
```

【使用范例】

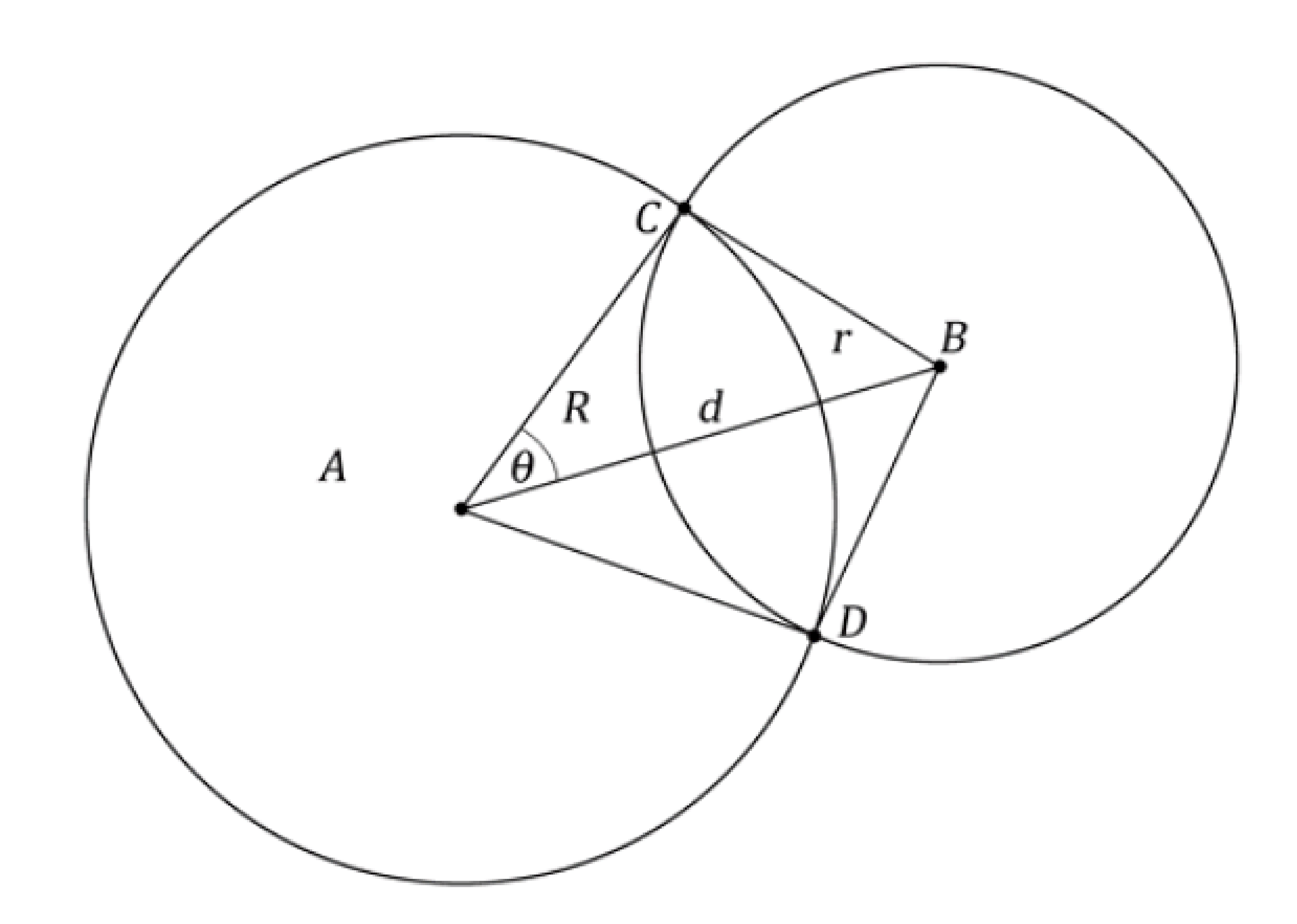
参见程序 ZOJ1450.CPP。

3.2.4 圆与圆求交

【任务】

求圆与圆的交点。

【说明】



如图所示，设$d = |AB|$。由余弦定理，$\cos\theta = (R^2 + d^2 - r^2)/(2Rd)$。显然$\theta$是锐角，所以可以计算出$\sin\theta = \sqrt{1 - \cos^2\theta}$。利用平面向量旋转公式，将向量$\overrightarrow{AB}$分别顺逆时针旋转$\theta$，并将长度伸缩到$R$，就可以得到两个向量$\overrightarrow{AD}$和$\overrightarrow{AC}$。然后很容易就可以求出$C$和$D$。虽然利用了角度，但是代码中只有一次除法运算和一次开根运算，并没有涉及三角函数的运算，不会引入很大的精度误差。

【接口】

point rotate(const point &p, double cost, double sint);

输入：$\&p$　　一个向量

　　　$cost, sint$　　旋转弧度θ的cos值和sin值

输出：将向量p旋转θ弧度得到的向量

pair<point, point> crosspoint(point ap, double ar, point bp, double br);

输入：ap, bp　　两个圆的圆心，分别对应上面的点A和点B

　　　ar, br　　两个圆的半径，分别对应上面的R和r

输出：圆A和B的两个交点

【代码】

```
point rotate(const point &p, double cost, double sint) {
    double x = p.x, y = p.y;
    return point(x*cost - y*sint, x*sint + y*cost);
}

pair<point, point> crosspoint(point ap,double ar,point bp,double br){
    double d = (ap - bp).norm();
```

```
    double cost = (ar*ar + d*d - br*br) / (2*ar*d);
    double sint = sqrt(1. - cost*cost);
    point v = (bp - ap) / (bp - ap).norm() * ar;
    return make_pair(ap+rotate(v,cost,-sint),ap+ rotate(v,cost,sint));
}
```

【注释】

请在调用crosspoint前确认两圆存在交点。

【使用范例】

参见程序 SPOJ_CIRU.CPP。

3.2.5 圆的离散化

【任务】

给定n个圆，求它们的面积并。

【说明】

将圆与圆之间的交点、圆的左右端点、以及圆心的x值离散出来作为事件点，并按照这些值，画竖直的线切割圆。这样相邻两条竖线之间，每个圆要么不在这里，要么有一上一下两个弧。这些弧之间除了在竖线上就再无交点。取每个弧的中点，排序后扫描即可。

getUnion中的cnt记录了当前区域被多少个圆所覆盖。因此可以通过适当的修改，求出恰好被k个圆覆盖的区域的面积。

【接口】

double getUnion(int n,Tcir a[ ]);

复杂度：$O(n^3 \log n)$，实际应用中一般远不到这个界

输　入：n　　圆的个数

　　　　a　　所有的圆

输　出：这些圆的面积并的面积

调用外部函数：

　圆与圆求交：参见 3.2.4 节

【代码】

```
const int maxn  =   55;
const int maxN  =   maxn*maxn+3*maxn;
struct Tcir
```

```
{
    double r;
    point o;
    void read(){scanf("%lf%lf%lf",&o.x,&o.y,&r);}
};
struct Tinterval
{
    double x,y,Area,mid;
    int type;
    Tcir owner;
    void area(double l,double r)
    {
        double len=sqrt(sqr(l-r) + sqr(x-y));
        double d=sqrt(sqr(owner.r)-sqr(len)/4.0);
        double angle=atan(len/2.0/d);
        Area=fabs(angle*sqr(owner.r)-d*len/2.0);
    }
}inter[maxn];
double x[maxN],l,r;
int n,N,Nn;

bool compR(const Tcir &a,const Tcir &b)
{
    return a.r>b.r;
}

void Get(Tcir owner,double x,double &l,double &r)
{
    double y=fabs(owner.o.x-x);
    double d=sqrt(fabs(sqr(owner.r) - sqr(y)));
    l=owner.o.y+d;
    r=owner.o.y-d;
}

void Get_Interval(Tcir owner,double l,double r)
{
    Get(owner,l,inter[Nn].x,inter[Nn+1].x);
    Get(owner,r,inter[Nn].y,inter[Nn+1].y);
    Get(owner,(l+r)/2.0,inter[Nn].mid,inter[Nn+1].mid);
```

```
    inter[Nn].owner=inter[Nn+1].owner=owner;
    inter[Nn].area(l,r);inter[Nn+1].area(l,r);
    inter[Nn].type=1;inter[Nn+1].type=-1;
    Nn+=2;
}

bool comp(const Tinterval &a,const Tinterval &b)
{
    return a.mid>b.mid+eps;
}

void Add(double xx)
{
    x[N++]=xx;
}

double dist2(const point &a,const point &b)
{
    return sqr(dist(a,b));
}

double getUnion(int n,Tcir a[])
{
    int p=0;
    sort(a,a+n,compR);
    for (int i=0;i<n;++i){
        bool fl=true;
        for (int j=0;j<i;++j)
        if (dist2(a[i].o,a[j].o)<=sqr(a[i].r-a[j].r)+1e-12){
            fl=false;
            break;
        }
        if (fl) a[p++]=a[i];
    }
    n=p;

    N=0;
    for (int i=0;i<n;++i){
        Add(a[i].o.x-a[i].r);
```

```
        Add(a[i].o.x+a[i].r);
        Add(a[i].o.x);
        for (int j=i+1;j<n;++j)
        if (dist2(a[i].o,a[j].o)<=sqr(a[i].r+a[j].r)+eps){
            pair<point,point> cross=crosspoint(a[i].o,a[i].r,a[j].o,
            a[j].r);
            Add(cross.first.x);
            Add(cross.second.x);
        }
    }
    sort(x,x+N);
    p=0;
    for (int i=0;i<N;++i)
    if (!i || fabs(x[i]-x[i-1])>eps) x[p++]=x[i];
    N=p;

    double ans=0;
    for (int i=0;i+1<N;++i){
        l=x[i],r=x[i+1];
        Nn=0;
        for (int j=0;j<n;++j)
        if (fabs(a[j].o.x-l)<a[j].r+eps && fabs(a[j].o.x-r)
        <a[j].r+eps)
            Get_Interval(a[j],l,r);
        if (Nn){
            sort(inter,inter+Nn,comp);
            int cnt=0;
            for (int i=0;i<Nn;++i){
                if (cnt>0){
                    ans+=(fabs(inter[i-1].x-inter[i].x)
                          +fabs(inter[i-1].y-inter[i].y))
                         *(r-l)/2.0;
                    ans+=inter[i-1].type*inter[i-1].Area;
                    ans-=inter[i].type*inter[i].Area;
                }
                cnt+=inter[i].type;
            }
        }
    }
```

```
    return ans;
}
```

【注释】

常数中，maxn为圆的个数，maxN为所有事件点的个数。

另外，新增了圆类Tcir和中间过程所需的区间类Tinterval。

【使用范例】

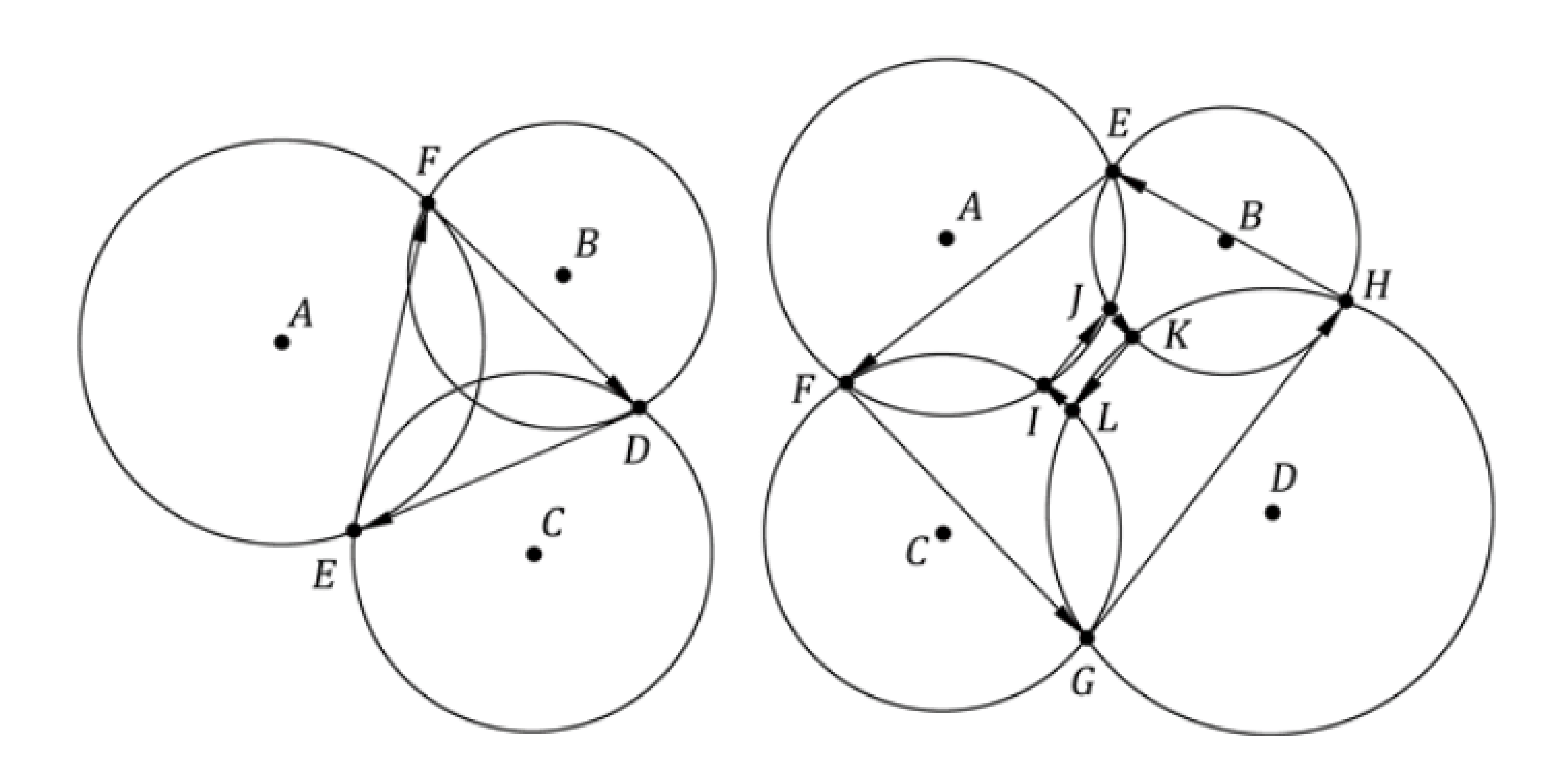
参见程序 SPOJ_VCIRCLE.CPP。

3.2.6　圆的面积并

【任务】

给定n个圆，求它们的面积并。

【说明】



如图所示，将圆的面积剖分成若干个多边形的面积与若干个弓形的面积。多边形的边就是圆的交点构成的不被其他圆覆盖的弦。计算有向面积的话，可以看到中间“洞”的面积恰好被顺时针的多边形包围，因此会被减去。请注意，必须去除重复的圆，否则答案会有重复计算的面积。

【接口】

double solve( );

复杂度：$O(m^2\log m)$

输　入：m　全局变量，圆的个数

　　　　tc　全局变量，待求面积并的圆

输　出：这些圆的面积并

调用外部函数：

圆与圆求交：参见 3.2.4 节

【代码】

```
double cross(const point &a, const point &b) {
    return a.x*b.y - a.y*b.x;
}

struct Circle {
    point p;
    double r;

    bool operator <(const Circle &o) const {
        if (dcmp(r - o.r) != 0) return dcmp(r - o.r) == -1;
        if (dcmp(p.x - o.p.x) != 0) {
            return dcmp(p.x - o.p.x) == -1;
        }
        return dcmp(p.y - o.p.y) == -1;
    }

    bool operator ==(const Circle &o) const {
        return dcmp(r - o.r) == 0 && dcmp(p.x - o.p.x == 0) &&
            dcmp(p.y - o.p.y) == 0;
    }
};

inline pair<point,point> crosspoint(const Circle &a,const Circle &b){
    return crosspoint(a.p, a.r, b.p, b.r);
}

Circle c[1000], tc[1000];
int n, m;

struct Node {
    point p;
    double a;
    int d;

    Node(const point &p, double a, int d) : p(p), a(a), d(d) {}
```

```
    bool operator <(const Node &o) const {
        return a < o.a;
    }
};

double arg(point p) {
    return arg(complex<double>(p.x, p.y));
}

double solve() {
    sort(tc, tc + m);
    m = unique(tc, tc + m) - tc;
    for (int i = m - 1; i >= 0; --i) {
        bool ok = true;
        for (int j = i + 1; j < m; ++j) {
            double d = (tc[i].p - tc[j].p).norm();
            if (dcmp(d - abs(tc[i].r - tc[j].r)) <= 0) {
                ok = false;
                break;
            }
        }
    if (ok) c[n++] = tc[i];
    }
    double ans = 0;
    for (int i = 0; i < n; ++i) {
        vector<Node> event;
        point boundary = c[i].p + point(-c[i].r, 0);
        event.push_back(Node(boundary, -PI, 0));
        event.push_back(Node(boundary, PI, 0));
        for (int j = 0; j < n; ++j) {
            if (i == j) continue;
            double d = (c[i].p - c[j].p).norm();
            if (dcmp(d - (c[i].r + c[j].r)) < 0) {
                pair<point, point> ret = crosspoint(c[i], c[j]);
                double x = arg(ret.first - c[i].p);
                double y = arg(ret.second - c[i].p);
                if (dcmp(x - y) > 0) {
                    event.push_back(Node(ret.first, x, 1));
```

```
                event.push_back(Node(boundary, PI, -1));
                event.push_back(Node(boundary, -PI, 1));
                event.push_back(Node(ret.second, y, -1));
            } else {
                event.push_back(Node(ret.first, x, 1));
                event.push_back(Node(ret.second, y, -1));
            }
        }
    }
    sort(event.begin(), event.end());
    int sum = event[0].d;
    for (int j = 1; j < (int)event.size(); ++j) {
        if (sum == 0) {
            ans += cross(event[j - 1].p, event[j].p) / 2;
            double x = event[j - 1].a;
            double y = event[j].a;
            double area = c[i].r * c[i].r * (y - x) / 2;
            point v1 = event[j - 1].p - c[i].p;
            point v2 = event[j].p - c[i].p;
            area -= cross(v1, v2) / 2;
            ans += area;
        }
        sum += event[j].d;
    }
  }
  return ans;
}
```

【使用范例】

参见程序 SPOJ_CIRU.CPP。

3.3 三维计算几何

3.3.1 三维点类

【任务】

完成三维点的基本操作，进行一些最基本的三维向量操作。

【说明】

参考代码注释。

【接口】

类：*Point_3*

成员变量：

double x, y, z　　点的坐标

重载运算符：+，−，×，/

成员函数：

Length( )　　计算向量的模长

Unit( )　　计算向量的单位向量

相关函数：

Point_3 Det(const Point_3 &a, const Point_3 &b);　　计算两个向量的叉积

double Dot(const Point_3 &a, const Point_3 &b);　　计算两个向量的点积

double Mix(const Point_3 &a, const Point_3 &b);　　计算两个向量的混合积

double dis(const Point_3 &a, const Point_3 &b);　　计算两个点的距离

【代码】

```
const double eps = 1e-8;
const double pi = acos(-1.0);
inline int cmp(double a) {
    return a < -eps ? -1 : a > eps;
}
inline double Sqr(double a) {
    return a * a;
}
inline double Sqrt(double a) {
    return a <= 0 ? 0 : sqrt(a);
}
Class Point_3 {
    Public:
    double x, y, z;
    Point_3() {
    }
    Point_3(double x, double y, double z) : x(x), y(y), z(z) {
    }
    //向量长度
```

```
    double Length() const {
    return Sqrt(Sqr(x) + Sqr(y) + Sqr(z));
    }
    Point_3 Unit() const;
};
Point_3 operator + (const Point_3 &a, const Point_3 &b) {
    return Point_3(a.x + b.x, a.y + b.y, a.z + b.z);
}
Point_3 operator - (const Point_3 &a, const Point_3 &b) {
    return Point_3(a.x - b.x, a.y - b.y, a.z - b.z);
}
Point_3 operator * (const Point_3 &a, double b) {
    return Point_3(a.x * b, a.y * b, a.z * b);
}
Point_3 operator / (const Point_3 &a, double b) {
    return Point_3(a.x / b, a.y / b, a.z / b);
}
//返回单位化的向量
Point_3 Point_3::Unit() const {
    return *this / Length();
}
//向量 a 和向量 b 的叉积，返回的是一个向量
Point_3 Det(const Point_3 &a, const Point_3 &b) {
    return Point_3(a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z, a.x *
            b.y - a.y * b.x);
}
//向量 a 和向量 b 的点积
double Dot(const Point_3 &a, const Point_3 &b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
//向量 a,b,c 的混合积。返回值除以 6 就是 a,b,c 这三个向量所构成的四面体的体积
double Mix(const Point_3 &a, const Point_3 &b, const Point_3 &c) {
    return Dot(a, Det(b, c));
}
//三维两点间距离
double dis(const Point_3 &a, const Point_3 &b){
     return Sqrt(Sqr(a.x-b.x) + Sqr(a.y-b.y) + Sqr(a.z-b.z));}
```

3.3.2 三维直线类

【任务】

完成三维线段与三维直线的基本操作。

【接口】

参见代码注释。

【代码】

```
class Line_3 {
    Public:
    Point_3 a,b;
    Line_3() {}
    Line_3(Point_3 a,Point_3 b) : a(a), b(b) {}
};
//线段长度
double vlen(Point_3 P) {return P.Length();}
//零值函数
bool zero(double x) { return fabs(x)<eps;}
//判断三点共线
int dots_inline(Point_3 p1,Point_3 p2,Point_3 p3){
     return vlen(det(p1-p2,p2-p3))<eps;}
//判断点在线段内（包含端点）
int dot_online_in(Point_3 p,Line_3l){
    return zero(vlen(det(p-l.a,p-l.b)))&&(l.a.x-p.x)*(l.b.x-p.x)<eps&&
           (l.a.y-p.y)*(l.b.y-p.y)<eps&&(l.a.z-p.z)*(l.b.z-p.z)<eps;}
//判断点在线段内（不包含端点）
int dot_online_ex(Point_3 p,Line_3l){
     return dot_online_in(p,l.a,l.b)&&(!zero(p.x-l.a.x)||!
     zero(p.y-l.a.y)
     ||!zero(p.z-l.a.z))&&(!zero(p.x-l.b.x)||!zero(p.y-l.b.y)||!
     zero(p.z-l.b.z));
}
//判断平面内两点在直线同侧
int same_side(Point_3 p1,Point_3 p2,Line_3 l){
     return dot(det(l.a-l.b,p1-l.b),det(l.a-l.b,p2-l.b))>eps;}
//判断平面内两点在直线异侧
int opposite_side(Point_3 p1,Point_3 p2,Line_3 l){
     return dot(det(l.a-l.b,p1-l.b), det(l.a-l.b,p2-l.b))<-eps;}
```

```
//判断两直线平行
int parallel(Line_3 u,Line_3 v){return vlen(det(u.a-u.b,v.a-v.b))
    <eps;}
//判断两直线垂直
int perpendicular(Line_3 u,Line_3 v){return zero(dot(u.a-u.b,
    v.a-v.b));}
//判断两条线段是否有交点（包含端点）
int intersect_in(Line_3 u,Line_3 v){
    if (!dots_onplane(u.a,u.b,v.a,v.b)) return 0;
    if (!dots_inline(u.a,u.b,v.a)||!dots_inline(u.a,u.b,v.b))
        return !same_side(u.a,u.b,v)&&!same_side(v.a,v.b,u);
    return dot_online_in(u.a,v)||dot_online_in(u.b,v)||
        dot_online_in(v.a,u)||dot_online_in(v.b,u);
}
//判断两条线段是否有交点（不包含端点）
int intersect_ex(Line_3 u,Line_3 v){
    return dots_onplane(u.a,u.b,v.a,v.b)&&opposite_side(u.a,u.b,v)&&
        opposite_side(v.a,v.b,u);
}
//求两直线交点（必须保证共面且不平行）
Point_3 intersection(Line_3 u,Line_3 v){
    Point_3 ret=u.a;
    double t=((u.a.x-v.a.x)*(v.a.y-v.b.y)-(u.a.y-v.a.y)*
            (v.a.x-v.b.x))
            /((u.a.x-u.b.x)*(v.a.y-v.b.y)-(u.a.y-u.b.y)*
            (v.a.x-v.b.x));
    ret+=(u.b-u.a)*t;   return ret;
}
//点到直线的距离
double ptoline(Point_3 p,Line_3 l){
    return vlen(det(p-l.a,l.b-l.a))/distance(l.a,l.b);}
//直线到直线的距离，平行时需特别处理
double linetoline(Line_3 u,Line_3 v){
    Point_3 n=det(u.a-u.b,v.a-v.b);
    return fabs(dot(u.a-v.a,n))/
           vlen(n);}
//求两直线夹角的 cos 值
double angle_cos(Line_3 u,Line_3 v){
    return dot(u.a-u.b,v.a-v.b)/vlen(u.a-u.b)/vlen(v.a-v.b);}
```

3.3.3 三维平面类

【任务】

完成三维平面的基本操作。

【接口】

参见代码注释。

【代码】

```
class Plane_3 {
    Public:
    Point_3 a,b,c;
    Plane_3() {}
    Plane_3(Point_3 a,Point_3 b,Point_3 c) : a(a), b(b), c(c) {}
};
//线段长度
double vlen(Point_3 P) {return P.Length();}
//零值函数
bool zero(double x) { return fabs(x)<eps;}
//平面法向量
Point_3 pvec(Point_3 s1,Point_3 s2,Point_3 s3){return det((s1-s2),
     (s2-s3));}
//判断四点共平面
int dots_onplane(Point_3 a,Point_3 b,Point_3 c,Point_3 d){
    return zero(dot(pvec(a,b,c),d-a));}
//判断一个点是否在三角形里(包含边界)
int dot_inplane_in(Point_3 p,Plane_3 s){
    return zero(vlen(det(s.a-s.b,s.a-s.c))-vlen(det(p-s.a,p-s.b))-
    vlen(det(p-s.b,p-s.c))-vlen(det(p-s.c,p-s.a)));}
//判断一个点是否在三角形里(不包含边界)
int dot_inplane_ex(Point_3 p,Plane_3 s){
    return dot_inplane_in(p,s.a,s.b,s.c)&&vlen(det(p-s.a,p-s.b))
    >eps&& vlen(det(p-s.b,p-s.c))>eps&&vlen(det(p-s.c,p-s.a))>eps;}
//判断两点在平面同侧
int same_side(Point_3 p1,Point_3 p2,Plane_3 s){
    return dot(pvec(s),p1-s.a)*dot(pvec(s),p2-s.a)>eps;}
//判断两点在平面异侧
int opposite_side(Point_3 p1,Point_3 p2,Plane_3 s){
    return dot(pvec(s),p1-s.a)*dot(pvec(s),p2-s.a)<-eps;}
```

```
//判断两平面平行
int parallel(Plane_3 u,Plane_3 v){return vlen(det(pvec(u),
    pvec(v)))<eps;}
//check if a plane and a line is parallel
int parallel(Line_3 l,Plane_3 s){ return zero(dot(l.a-l.b,pvec(s))); }
//判断两平面垂直
int perpendicular(Plane_3 u,Plane_3 v){return zero(dot(pvec(u),
    pvec(v))); }
//判断直线与平面垂直
int perpendicular(Line_3 l,Plane_3 s){return vlen(det(l.a-l.b,
    pvec(s)))<eps;}
//判断线段和三角形是否有交点(包含边界)
int intersect_in(Line_3 l,Plane_3 s){
    return !same_side(l.a,l.b,s)&&
           !same_side(s.a,s.b,Plane_3(l.a,l.b,s.c))&&
           !same_side(s.b,s.c,Plane_3(l.a,l.b,s.a))&&
           !same_side(s.c,s.a,Plane_3(l.a,l.b,s.b));
}
//判断线段和三角形是否有交点(不包含边界)
int intersect_ex(Line_3 l,Plane_3 s){
    return opposite_side(l.a,l.b,s) &&
           opposite_side(s.a,s.b,Plane_3(l.a,l.b,s.c))&&
           opposite_side(s.b,s.c,Plane_3(l.a,l.b,s.a))&&
           opposite_side(s.c,s.a,Plane_3(l.a,l.b,s.b));}
//求直线与平面的交点
Point_3 intersection(Line_3 l,Plane_3 s){
    Point_3 ret=pvec(s);
    double t=(ret.x*(s.a.x-l.a.x)+ret.y*(s.a.y-l.a.y)+ret.z*(s.a.z-
             l.a.z))/
             (ret.x*(l.b.x-l.a.x)+ret.y*(l.b.y-l.a.y)+ret.z*
             (l.b.z-l.a.z));
    ret=l.a + (l.b-l.a)*t; return ret;
}
//求两平面的交线
bool intersection(Plane_3 pl1 , Plane_3 pl2 , Line_3 &li) {
    if (parallel(pl1,pl2)) return false;
    li.a=parallel(pl2.a,pl2.b, pl1) ? intersection(pl2.b,pl2.c,
         Plane_3(pl1.a,pl1.b,pl1.c)):
    intersection(pl2.a,pl2.b,Plane_3(pl1.a,pl1.b,pl1.c));
```

```
        Point_3 fa;fa=det(pvec(pl1),pvec(pl2));li.b=li.a+fa;return true;
}
//点到平面的距离
double ptoplane(Point_3 p,Plane_3 s){
        return fabs(dot(pvec(s),p-s.a))/vlen(pvec(s));}
//求两平面的夹角的 cos 值
double angle_cos(Plane_3 u,Plane_3 v){
        return dot(pvec(u),pvec(v))/vlen(pvec(u))/vlen(pvec(v));}
//求平面与直线的夹角的 sin 值
double angle_sin(Line_3 l,Plane_3 s){
        return dot(l.a-l.b,pvec(s))/vlen(l.a-l.b)/vlen(pvec(s));}
```

3.3.4 三维向量旋转

【任务】

给定a,b两点和$angle$，将a绕Ob向量逆时针旋转弧度$angle$。

【说明】

求出Ob单位方向向量e_3，利用点乘，求出Oa在Ob上的投影点p，设pa的单位向量设为e_1，利用e_1叉乘e_3求出单位法向量e_2，求出a在e_1,e_2上的投影x_1,y_1。

将x_1,y_1旋转 $angle$ 度。即：

$$x = x_1 \times \cos(angle) - y_1 \times \sin(angle)$$
$$y = x_1 \times \sin(angle) + y_1 \times \cos(angle)$$

新坐标即为$e1 \times y + e2 \times x + p$。

【接口】

point rotate(point a,point b,double angle);

复杂度：O(1)

输　入：a,b　　同任务描述

　　　　$angle$　　旋转的弧度

输　出：将a绕Ob向量逆时针旋转弧度$angle$的结果

【代码】

```
point rotate(point a, point b, double angle){
    point e1,e2,e3;
    e3=b.std();
    double len=dot(a,e3);
```

```
    point p=e3*len;
    e1=a-p;
    if (e1.len()>(1e-8)) e1.std();
    e2=cross(e1,e3);
    double x1=dot(a,e1), y1=dot(a,e2);
    x=x1*cos(angle)-y1*sin(angle);
    y=x1*sin(angle)+y1*cos(angle);
    return e1*y+e2*x+p;
}
```

【使用范例】

参见程序 POJ3391.CPP。

3.3.5 长方体表面两点最短距离

【任务】

给出长方体上两点坐标，求其在长方体表面上的最短路径。

【说明】

将长方体旋转，使得有一个点在底面上。

枚举该点到另外一点的最短路径经过哪些平面，然后将平面铺平后处理。

枚举的方法为：枚举下一个面是当前面的上、下、左、右面，turn中的i、j分别表示在竖直方向和水平方向上各经过了几个面（正负表示方向）。需要注意的是最优解中若已经向右转过，则不可能再向左转。

【接口】

int rect_dist(int L,int W,int H,int x1,int y1,int z1,int x2,int y2,int z2);

复杂度：O(1)

输　入：L,W,H　　当前长方体的各边长

$x1,y1,z1,x2,y2,z2$　　两点坐标

输　出：长方体表面上的最短路径的平方

【代码】

```
int ans;
void turn(int i,int j,int x,int y,int z,int x0,int y0,int L,int W,int H)
{
    if (z==0)
```

```
            ans=min(ans,x*x+y*y);
        else
        {
            if (i>=0 && i<2)
                turn(i+1,j,x0+L+z,y,x0+L-x,x0+L,y0,H,W,L);
            if (j>=0 && j<2)
                turn(i,j+1,x,y0+W+z,y0+W-y,x0,y0+W,L,H,W);
            if (i<=0 && i>-2)
                turn(i-1,j,x0-z,y,x-x0,x0-H,y0,H,W,L);
            if (j<=0 && j>-2)
                turn(i,j-1,x,y0-z,y-y0,x0,y0-H,L,H,W);
         }
}
int rect_dist(int L,int W,int H,int x1,int y1,int z1,int x2,int y2,
int z2)
{
    if (z1!=0 && z1!=H)
        if (y1==0 || y1==W)
            swap(y1,z1),swap(y2,z2),swap(W,H);
        else
            swap(x1,z1),swap(x2,z2),swap(L,H);
    if (z1==H)
        z1=0,z2=H-z2;
    ans=1<<30;
    turn(0,0,x2-x1,y2-y1,z2,-x1,-y1,L,W,H);
    return(ans);
}
```

【使用范例】

参见程序 POJ2977.CPP。

3.3.6 四面体体积

【任务】

给定四面体六条棱的长度，计算此四面体的体积。

【说明】

下面的推导过程是由欧拉提出来的，又称欧拉四面体体积公式。

（1）建立x,y,z直角坐标系。设A,B,C三点的坐标分别为(a_1,b_1,c_1)，(a_2,b_2,c_2)，(a_3,b_3,c_3)，四面体$O-ABC$的六条棱长分别为l,m,n,p,q,r；

（2）四面体的体积为

$$V=\frac{1}{6}(\overrightarrow{OA}\times\overrightarrow{OB})\cdot\overrightarrow{OC}=\frac{1}{6}\begin{vmatrix}a_1 & b_1 & c_1\\ a_2 & b_2 & c_2\\ a_3 & b_3 & c_3\end{vmatrix}$$

将这个式子平方后得到：

$$V^2=\frac{1}{36}\begin{vmatrix}a_1^2+b_1^2+c_1^2 & a_1a_2+b_1b_2+c_1c_2 & a_1a_3+b_1b_3+c_1c_3\\ a_1a_2+b_1b_2+c_1c_2 & a_2^2+b_2^2+c_2^2 & a_2a_3+b_2b_3+c_2c_3\\ a_1a_3+b_1b_3+c_1c_3 & a_2a_3+b_2b_3+c_2c_3 & a_3^2+b_3^2+c_3^2\end{vmatrix}(*)$$

（3）根据矢量数量积的坐标表达式及数量积的定义得

$$a_1^2+b_1^2+c_1^2=\overrightarrow{OA}\cdot\overrightarrow{OA}=\left|\overrightarrow{OA}\right|^2=p^2$$
$$a_2^2+b_2^2+c_2^2=\overrightarrow{OB}\cdot\overrightarrow{OB}=\left|\overrightarrow{OB}\right|^2=q^2$$
$$a_3^2+b_3^2+c_3^2=\overrightarrow{OC}\cdot\overrightarrow{OC}=\left|\overrightarrow{OC}\right|^2=r^2$$

又根据余弦定理得：

$$a_1a_2+b_1b_2+c_1c_2=\overrightarrow{OA}\cdot\overrightarrow{OB}=p\cdot q\cdot cos(p,q)=\frac{p^2+q^2-n^2}{2}$$
$$a_1a_3+b_1b_3+c_1c_3=\overrightarrow{OA}\cdot\overrightarrow{OC}=p\cdot r\cdot cos(p,r)=\frac{p^2+r^2-m^2}{2}$$
$$a_2a_3+b_2b_3+c_2c_3=\overrightarrow{OB}\cdot\overrightarrow{OC}=q\cdot r\cdot cos(q,r)=\frac{q^2+r^2-l^2}{2}$$

（4）将上述的式子带入(*)，就得到了欧拉四面体体积公式

$$V^2=\frac{1}{36}\begin{vmatrix}p^2 & \frac{p^2+q^2-n^2}{2} & \frac{p^2+r^2-m^2}{2}\\ \frac{p^2+q^2-n^2}{2} & q^2 & \frac{q^2+r^2-l^2}{2}\\ \frac{p^2+r^2-m^2}{2} & \frac{q^2+r^2-l^2}{2} & r^2\end{vmatrix}$$

【接口】

double volume(double l,double n,double a,double m,double b,double c)

复杂度：O(1)

输　入：l,n,a,m,b,c　四面体六条棱的长度

输　出：四面体的体积

对于四面体$O-ABC$，调用calc(OA,OB,OC,AB,AC,BC)即得体积。

【代码】

```
double volume(double l,double n,double a,double m,double b,double c){
```

```
double x,y;
    x=4*a*a*b*b*c*c-a*a*(b*b+c*c-m*m)*(b*b+c*c-m*m)-
         b*b*(c*c+a*a-n*n)*(c*c+a*a-n*n);
    y=c*c*(a*a+b*b-l*l)*(a*a+b*b-l*l)-(a*a+b*b-l*l)
         *(b*b+c*c-m*m)*(c*c+a*a-n*n);
    return(sqrt(x-y)/12);
}
```

【使用范例】

参见程序 POJ2208.CPP。

3.3.7 最小球覆盖

【任务】

要求一个半径最小的球覆盖住所有的点。

【说明】

类似最小圆覆盖的算法，需要写分别由 1~4 个点确定一个球的过程。

【接口】

void ball( );

复杂度：$O(n)$

输　入：	*npoint*	全局变量，点数
	pt	全局变量，点的坐标
输　出：	*res*	全局变量，球心坐标
	radius	全局变量，球的半径

【代码】

```
int npoint, nouter;
Tpoint pt[200000], outer[4],res;
double radius,tmp;

inline double dist(Tpoint p1, Tpoint p2) {
    double dx=p1.x-p2.x, dy=p1.y-p2.y, dz=p1.z-p2.z;
    return (dx*dx + dy*dy + dz*dz);
}

inline double dot(Tpoint p1, Tpoint p2) {
```

```
    return p1.x*p2.x + p1.y*p2.y + p1.z*p2.z;
}

void ball() {
    Tpoint q[3]; double m[3][3], sol[3], L[3], det;
    int i,j;
    res.x = res.y = res.z = radius = 0;
    switch (nouter) {
        case 1: res=outer[0]; break;
        case 2:
            res.x=(outer[0].x+outer[1].x)/2;
            res.y=(outer[0].y+outer[1].y)/2;
            res.z=(outer[0].z+outer[1].z)/2;
            radius=dist(res, outer[0]);
            break;
        case 3:
            for (i=0; i<2; ++i) {
                q[i].x=outer[i+1].x-outer[0].x;
                q[i].y=outer[i+1].y-outer[0].y;
                q[i].z=outer[i+1].z-outer[0].z;
            }
            for (i=0; i<2; ++i) for(j=0; j<2; ++j)
                m[i][j]=dot(q[i], q[j])*2;
            for (i=0; i<2; ++i) sol[i]=dot(q[i], q[i]);
            if (fabs(det=m[0][0]*m[1][1]-m[0][1]*m[1][0])<eps)
                return;
            L[0]=(sol[0]*m[1][1]-sol[1]*m[0][1])/det;
            L[1]=(sol[1]*m[0][0]-sol[0]*m[1][0])/det;
            res.x=outer[0].x+q[0].x*L[0]+q[1].x*L[1];
            res.y=outer[0].y+q[0].y*L[0]+q[1].y*L[1];
            res.z=outer[0].z+q[0].z*L[0]+q[1].z*L[1];
            radius=dist(res, outer[0]);
            break;
        case 4:
            for (i=0; i<3; ++i) {
                q[i].x=outer[i+1].x-outer[0].x;
                q[i].y=outer[i+1].y-outer[0].y;
                q[i].z=outer[i+1].z-outer[0].z;
                sol[i]=dot(q[i], q[i]);
```

```
                }
                for (i=0;i<3;++i)
                    for(j=0;j<3;++j) m[i][j]=dot(q[i],q[j])*2;
                det= m[0][0]*m[1][1]*m[2][2]
                    + m[0][1]*m[1][2]*m[2][0]
                    + m[0][2]*m[2][1]*m[1][0]
                    - m[0][2]*m[1][1]*m[2][0]
                    - m[0][1]*m[1][0]*m[2][2]
                    - m[0][0]*m[1][2]*m[2][1];
                if (fabs(det)<eps) return;
                for (j=0; j<3; ++j) {
                    for (i=0; i<3; ++i) m[i][j]=sol[i];
                    L[j]=(m[0][0]*m[1][1]*m[2][2]
                          + m[0][1]*m[1][2]*m[2][0]
                          + m[0][2]*m[2][1]*m[1][0]
                          - m[0][2]*m[1][1]*m[2][0]
                          - m[0][1]*m[1][0]*m[2][2]
                          - m[0][0]*m[1][2]*m[2][1]
                          ) / det;
                    for (i=0; i<3; ++i)
                    m[i][j]=dot(q[i], q[j])*2;
            }
            res=outer[0];
            for (i=0; i<3; ++i) {
                res.x += q[i].x * L[i];
                res.y += q[i].y * L[i];
                res.z += q[i].z * L[i];
            }
            radius=dist(res, outer[0]);
        }
}

void minball(int n) {
    ball();
    if (nouter<4)
        for (int i=0; i<n; ++i)
            if (dist(res, pt[i])-radius>eps) {
                outer[nouter]=pt[i];
                ++nouter;
```

```
                minball(i);
                --nouter;
                if (i>0) {
                    Tpoint Tt = pt[i];
                    memmove(&pt[1], &pt[0], sizeof(Tpoint)*i);
                    pt[0]=Tt;
                }
            }
}
double smallest_ball(){
    radius=-1;
    for (int i=0;i<npoint;i++){
        if (dist(res,pt[i])-radius>eps){
            nouter=1;
            outer[0]=pt[i];
            minball(i);
        }
    }
    return sqrt(radius);
}
```

【使用范例】

参见程序 POJ2069.CPP。

3.3.8　三维凸包

【任务】

给出空间中上n个点，求出这n个点所构成的三维凸包的表面积。

【说明】

这里使用的是随机增量法。首先将输入的点打乱顺序，然后选择四个不共面的点组成一个小的四面体，如果找不到，则凸包不存在。否则，每次加入一个点，不断更新当前的凸包即可。更新的方法是：

（1）如果当前点已经在凸包内，则不需要更新；

（2）如果当前点在凸包之外，那么找到所有这样的原凸包上的边：过这条边的两个面一个可以被当前点看到，另一个不能。以这三个点新建一个面加入凸包中，这样就得到了一个包含当前所有点的新的凸包。

【接口】

double 3D_convex( );

复杂度：$O(n^2)$

输　入：*info*　全局变量，读入的所有点

输　出：凸包的表面积

【代码】

```
#define SIZE(X) (int(X.size()))
#define PI 3.14159265358979323846264338327950288

const double eps = 1e-8;

inline int Sign(double x) {
    return x < -eps ? -1 : (x > eps ? 1 : 0);
}
inline double Sqrt(double x) {
    return x < 0 ? 0 : sqrt(x);
}

int mark[1005][1005];
Point_3 info[1005];
int n, cnt;

double mix(const Point_3 &a, const Point_3 &b, const Point_3 &c) {
    return a.dot(b.cross(c));
}
double area(int a, int b, int c) {
    return ((info[b] - info[a]).cross(info[c] - info[a])).length();
}
double volume(int a, int b, int c, int d) {
    return mix(info[b] - info[a], info[c] - info[a], info[d] - info[a]);
}
struct Face {
    int a, b, c;
    Face() {}
    Face(int a, int b, int c): a(a), b(b), c(c) {}
    int &operator [](int k) {
        if (k == 0) return a;
        if (k == 1) return b;
```

```
        return c;
    }
};

vector <Face> face;

inline void insert(int a, int b, int c) {
    face.push_back(Face(a, b, c));
}
void add(int v) {
    vector <Face> tmp;
    int a, b, c;
    cnt++;
    for (int i = 0; i < SIZE(face); i++) {
        a = face[i][0];
        b = face[i][1];
        c = face[i][2];
        if (Sign(volume(v, a, b, c)) < 0)
            mark[a][b] = mark[b][a] = mark[b][c] = mark[c][b]
            = mark[c][a] = mark[a][c] = cnt;
        else
            tmp.push_back(face[i]);
    }
    face = tmp;
    for (int i = 0; i < SIZE(tmp); i++) {
        a = face[i][0];
        b = face[i][1];
        c = face[i][2];
        if (mark[a][b] == cnt) insert(b, a, v);
        if (mark[b][c] == cnt) insert(c, b, v);
        if (mark[c][a] == cnt) insert(a, c, v);
    }
}

int Find() {
    for (int i = 2; i < n; i++) {
        Point_3 ndir = (info[0] - info[i]).cross(info[1] - info[i]);
        if (ndir == Point_3()) continue;
        swap(info[i], info[2]);
        for (int j = i + 1; j < n; j++)
```

```
            if (Sign(volume(0, 1, 2, j)) != 0) {
                swap(info[j], info[3]);
                insert(0, 1, 2);
                insert(0, 2, 1);
                return 1;
            }
    }
    return 0;
}

double 3D_convex() {
    sort(info, info + n);
    n = unique(info, info + n) - info;
    face.clear();
    random_shuffle(info, info + n);
    if (Find()) {
        memset(mark, 0, sizeof(mark));
        cnt = 0;
        for (int i = 3; i < n; i++) add(i);
        double ans = 0;
        for (int i = 0; i < SIZE(face); ++i) {
            Point_3 p = (info[face[i][0]] - info[face[i][1]]).cross
                        (info[face[i][2]] - info[face[i][1]]);
            ans += p.length();
        }
        return ans / 2.;
    }
    return -1; // no solution
}
```

【使用范例】

参见程序 POJ3528.CPP。

3.4 其　　他

3.4.1 三角形的四心

【任务】

计算三角形的重心、垂心、内心、外心坐标。

【说明】

重心：三点坐标平均即可；

外心：套用三点确定圆的程序；

垂心：根据外心、重心与垂心的关系（欧拉定理）可得；

内心：三点坐标按对边长度加权平均。

【接口】

point Triangle_Mass_Center(point a,point b,point c);

point CircumCenter (point a,point b,point c);

point Orthocenter(point a, point b, point c);

point Innercenter(point a, point b, point c);

输入：*a*, *b*, *c*　三角形的三个顶点

输出：一个点，分别表示三角心的重心、外心、垂心、内心

【代码】

```
point Triangle_Mass_Center(point a,point b,point c){
      return (a+b+c)/3.;
}
point CircumCenter (point a,point b,point c){
     point cp;
     double a1=b.x-a.x, b1=b.y-a.y, c1=(a1*a1+b1*b1)/2;
     double a2=c.x-a.x, b2=c.y-a.y, c2=(a2*a2+b2*b2)/2;
     double d=a1*b2-a2*b1 ;
     cp.x=a.x+(c1*b2-c2*b1)/d;
     cp.y=a.y+(a1*c2-a2*c1)/d;
     return cp;
}
point Orthocenter(point a, point b, point c){
      return Triangle_Mass_Center(a,b,c)*3.0-CircumCenter(a,b,c)*2.0;
}
point Innercenter(point a, point b, point c){
      point cp;
      double la,lb,lc;
      la=(b-c).norm();
      lb=(c-a).norm();
      lc=(a-b).norm();
      cp.x=(la*a.x+lb*b.x+lc*c.x)/(la+lb+lc);
```

```
    cp.y=(la*a.y+lb*b.y+lc*c.y)/(la+lb+lc);
    return cp;
}
```

【使用范例】

参见程序 POJ1673.CPP，ZOJ1776.CPP。

3.4.2 最近点对

【任务】

给出平面上n个点，求最近的两点间的距离。

【说明】

将n个点按x坐标进行排序，分为左右两半，分别求出两半内的最近点对的距离，取最小值作为当前答案，设为ans。然后考虑把两个点集合并，在分割线处还有可能会有更优的答案。取分割线左右ans距离内的点，按y坐标排序，用每个点与其左右6个点的距离更新答案即可。

【接口】

double Min_Dist(point a[], int s[], int n);

复杂度：O($n$log$n$)

输　入：n　点数

　　　　a　所有点的坐标

　　　　s　所有点按x坐标排序后的编号

输　出：最近点对的距离

【代码】

```
const int maxn=100000;                  //最大点数
point a[maxn];
int n,s[maxn];

bool cmpx(int i,int j){
    return cmp(a[i].x-a[j].x)<0;
}
bool cmpy(int i,int j){
    return cmp(a[i].y-a[j].y)<0;
}
```

```
double min_dist(point a[], int s[], int l,int r){
    double ans=1e100;
    if (r-l<20){
        for (int q=l;q<r;q++)
        for (int w=q+1;w<r;w++) ans=min(ans,(a[s[q]]-a[s[w]]).norm());
        return ans;
    }
    int tl,tr,m=(l+r)/2;
    ans=min(min_dist(a,s,l,m),min_dist(a,s,m,r));
    for (tl=l;a[s[tl]].x<a[s[m]].x-ans;tl++);
    for (tr=r-1;a[s[tr]].x>a[s[m]].x+ans;tr--);
    sort(s+tl,s+tr,cmpy);
    for (int q=tl;q<tr;q++)
       for (int w=q+1;w<min(tr,q+6);w++)
           ans=min(ans,(a[s[q]]-a[s[w]]).norm());
    sort(s+tl,s+tr,cmpx);
    return ans;
}
double Min_Dist(point a[], int s[], int n) {
    for (int i=0;i<n;i++) s[i]=i;
    sort(s,s+n,cmpx);
    return min_dist(a,s,0,n);
}
```

【使用范例】

参见程序 ZOJ2107.CPP。

3.4.3 平面最小曼哈顿距离生成树

【任务】

给定n个二维平面的点的坐标，两点间的距离为曼哈顿距离，求最小生成树。

【说明】

对于每个点，最小生成树上的边只可能是它与以它为原点将平面划分成8块（45°角）的每块中距离最近的点之间的边。

应用这个结论，又由于无向边的关系，因此只需要考虑其中的四块。对于每一块，相当于要查询一个三角形区域内的最小值，这个操作可以通过按其中一个限制排序，用数据结构（如树状数组）来完成另一限制下的查询，就可以求出那些可能的边。最后用这些可

能的边计算最小生成树即可。

【接口】

long long MinimumManhattanSpaningTree(int x[ ], int y[ ], int n);

复杂度：O($n$log$n$)

输　入：n　　点的个数

　　　　x, y　　n个点的x以及y坐标

输　出：这些点的最小生成树的权值和

Process(x,id,n)是一个离散化过程，将一个大小为n的数组x离散化后，将标号记录到id数组。cmp1,cmp2,cmp3,cmp4是比较函数。get_min和insert是树状数组过程。

【代码】

```
const int MAXN = 111111;
const int INF = 0x3fffffff;

inline int lowbit(const int &x) { return x & -x; }

struct Edge {
    int u, v, c;
    Edge(int _u = 0, int _v = 0, int _c = 0) : u(_u), v(_v), c(_c) {}
} edge[MAXN * 4];
inline bool operator< (const Edge &a, const Edge &b) { return a.c < b.c; }

struct Node {
    int key, id;
    Node(int _k = 0, int _i = 0) : key(_k), id(_i) {}
} Tree[MAXN];
inline bool operator < (const Node &a, const Node &b)
{ return a.key < b.key; }

int IDx[MAXN], IDy[MAXN], bak[MAXN]
int x[MAXN], y[MAXN], id[MAXN], father[MAXN];

int find(const int &x) {
    return father[x] == x ? x : father[x] = find(father[x]);
}

inline bool cmp1(const int &i, const int &j) {
```

```
    return x[i] - y[i] > x[j] - y[j] || x[i] - y[i] == x[j] - y[j] &&
    y[i] > y[j];
}
inline bool cmp2(const int &i, const int &j) {
    return x[i] - y[i] < x[j] - y[j] || x[i] - y[i] == x[j] - y[j]&&
    y[i] > y[j];
}
inline bool cmp3(const int &i, const int &j) {
    return x[i] + y[i] < x[j] + y[j] || x[i] + y[i] == x[j] + y[j] &&
    y[i] > y[j];
}
inline bool cmp4(const int &i, const int &j) {
    return x[i] + y[i] < x[j] + y[j] || x[i] + y[i] == x[j] + y[j] &&
    y[i] < y[j];
}

inline void Process(int x[], int idx[], int n) {
    for (int i = 0; i < n; ++i)
        bak[i] = x[i];
    sort(bak, bak + n, greater<int>());
    int p = unique(bak, bak + n) - bak;
    for (int i = 0; i < n; ++i)
        idx[i] = lower_bound(bak, bak + p, x[i], greater<int>())
        - bak + 1;
}

inline void add_edge(int &N, const int &u, const int &v) {
    edge[N++] = Edge(u, v, abs(x[u] - x[v]) + abs(y[u] - y[v]));
}

inline int get_min(const int &p) {
    Node tmp(INF);
    for (int i = p; i; i ^= lowbit(i))
        if (Tree[i].id != -1)
            tmp = min(tmp, Tree[i]);
    return tmp.key == INF ? -1 : tmp.id;
}

inline void insert(const int &n, const int &p, const Node &it) {
```

```
    for (int i = p; i <= n; i += lowbit(i))
        if (Tree[i].id == -1 || it < Tree[i])
            Tree[i] = it;
}

inline long long MinimumManhattanSpaningTree(int x[], int y[], int n){
    Process(x, IDx, n);
    Process(y, IDy, n);
    int N = 0;
    for (int i = 0; i < n; ++i)
        id[i] = i;
    sort(id, id + n, cmp1);
    for (int i = 1; i <= n; ++i)
        Tree[i].id = -1;
    for (int i = 0; i < n; ++i) {
        int u = id[i], v = get_min(IDy[u]);
        if (v != -1)
            add_edge(N, u, v);
        insert(n, IDy[u], Node(x[u] + y[u], u));
    }
    for (int i = 0; i < n; ++i)
        id[i] = i;
    sort(id, id + n, cmp2);
    for (int i = 1; i <= n; ++i)
        Tree[i].id = -1;
    for (int i = 0; i < n; ++i) {
        int u = id[i], v = get_min(IDx[u]);
        if (v != -1)
            add_edge(N, u, v);
        insert(n, IDx[u], Node(x[u] + y[u], u));
    }
    for (int i = 0; i < n; ++i)
        id[i] = i;
    sort(id, id + n, cmp3);
    for (int i = 1; i <= n; ++i)
        Tree[i].id = -1;
    for (int i = 0; i < n; ++i) {
        int u = id[i], v = get_min(IDy[u]);
        if (v != -1)
```

```
            add_edge(N, u, v);
        insert(n, IDy[u], Node(-x[u] + y[u], u));
    }
    for (int i = 0; i < n; ++i)
        id[i] = i;
    sort(id, id + n, cmp4);
    for (int i = 1; i <= n; ++i)
        Tree[i].id = -1;
    for (int i = 0; i < n; ++i) {
        int u = id[i], v = get_min(IDx[u]);
        if (v != -1)
            add_edge(N, u, v);
        insert(n, IDx[u], Node(x[u] - y[u], u));
    }
    //Kruskal
    sort(edge, edge + N);
    for (int i = 0; i < n; ++i)
        father[i] = i;
    long long res = 0;
    for (int i = 0; i < N; ++i) {
        int u = find(edge[i].u), v = find(edge[i].v);
        if (u != v) {
            father[u] = v;
            res += edge[i].c;
        }
    }
    return res;
}
```

【使用范例】

参见程序 BEIJING2006C.CPP。

3.4.4　最大空凸包

【任务】

给定n个点，求最大空凸包。

【说明】

穷举所要求解的空凸包的最低最左点（先保证最低，再保证最左）。

对于每一个穷举到的点v，进行动态规划，用$opt[i][j]$表示符合如下限制的凸包中的最大面积：

在凸包上v顺时针过来第一个点是i，并且i顺时针过来第一个点k不在$i \to j$的左手域（k可以等于j）。

【接口】

double Empty( );

复杂度：O(n^3)

输　入：n　　全局变量，表示点数

　　　　dot　　全局变量，所有点的坐标

输　出：最大空凸包的大小

【代码】

```
const int maxn = 100;
const double zero = 1e-8;

struct Vector {
    double x, y;
};

inline Vector operator - (Vector a, Vector b) {
    Vector c;
    c.x = a.x - b.x;
    c.y = a.y - b.y;
    return c;
}

inline double Sqr(double a) {
    return a * a;
}

inline int Sign(double a) {
    if (fabs(a) <= zero) return 0;
    return a < 0 ? -1 : 1;
}

inline bool operator < (Vector a, Vector b) {
    return Sign(b.y - a.y) > 0 || Sign(b.y - a.y) == 0 && Sign(b.x - a.x)>0;
```

```
}

inline double Max(double a, double b) {
    return a > b ? a : b;
}

inline double Length(Vector a) {
    return sqrt(Sqr(a.x) + Sqr(a.y));
}

inline double Cross(Vector a, Vector b) {
    return a.x * b.y - a.y * b.x;
}

Vector dot[maxn], List[maxn];
double opt[maxn][maxn];
int seq[maxn];
int n, len;
double ans;

bool Compare(Vector a, Vector b) {
    int temp = Sign(Cross(a, b));
    if (temp != 0) return temp > 0;
    temp = Sign(Length(b) - Length(a));
    return temp > 0;
}

void Solve(int vv) {
    int t, i, j, _len;
    for (i = len = 0; i < n; i++)
        if (dot[vv] < dot[i]) List[len++] = dot[i] - dot[vv];
    for (i = 0; i < len; i++)
        for (j = 0; j < len; j++)
            opt[i][j] = 0;
    sort(List, List + len, Compare);
    double v;
    for (t = 1; t < len; t++) {
        _len = 0;
        for (i = t - 1; i >= 0 && Sign(Cross(List[t], List[i])) == 0; i--);
```

```
        while (i >= 0) {
            v = Cross(List[i], List[t]) / 2;
            seq[_len++] = i;
            for (j = i - 1; j >= 0 && Sign(Cross(List[i] - List[t],
            List[j] - List[t])) > 0;j--);
            if (j >= 0) v += opt[i][j];
            ans = Max(ans, v);
            opt[t][i] = v;
            i = j;
        }
        for (i = _len - 2; i >= 0; i--)
            opt[t][seq[i]] = Max(opt[t][seq[i]], opt[t][seq[i + 1]]);
    }
}

int i;

double Empty() {
    ans = 0;
    for (i = 0; i < n; i++)
        Solve(i);
    return ans;
}
```

【使用范例】

参见程序 POJ1259.CPP。

3.4.5 平面划分

【任务】

给定n条直线，求出所有有限区域的面积。

【说明】

将所有交点求出后，将相邻的点之间进行连边，并且将这样的边拆成来回两条有向边。之后对每条还没有被遍历过的边进行遍历，每次找一条逆时针转角最小的边继续遍历，如此操作每次挖出来的正是一个简单区域，直接统计面积即可。对于最大的一块区域，则是最外层的那块无限区域，所以要删除。如果要将区域记下或是得到别的类似的信息，只要稍作修改即可。更具体的操作见程序。

【接口】

vector<double> Divide( );

复杂度：O(N^4)

输　入：N　　全局变量，直线条数

a,b　　全局变量，$a[i],b[i]$表示第i条直线上的两个不同的点

输　出：划分后每一个有限区域的面积

【代码】

```
#define SIZE(X) ((int)(X.size()))
#define PB push_back
#define MP make_pair

typedef pair<double, double> point;
#define X first
#define Y second

const double eps = 1e-8;
const double pi = acos(-1.);
const int maxm = 200000;
const int maxp = 20000;
const int maxn = 90;

int e[maxm], prev[maxm], mark[maxm], tote;
int info[maxp];

int N, P;
point a[maxn], b[maxn];
point p[maxp];

bool zero(double x) {
    return fabs(x) < eps;
}
point operator -(const point &a, const point &b) {
    return MP(a.X - b.X, a.Y - b.Y);
}
point operator *(const point &a, double k) {
    return point(a.X * k, a.Y * k);
}
```

```
point operator /(const point &a, double k) {
    return point(a.X / k, a.Y / k);
}
double getAngle(const point &a) {
    return atan2(a.Y, a.X);
}
double det(const point &a, const point &b) {
    return a.X * b.Y - a.Y * b.X;
}
bool operator ==(const point &a, const point &b) {
    return zero(a.X - b.X) && zero(a.Y - b.Y);
}

bool intersect(const point &a, const point &b, const point &c,
               const point &d, point &res) {
    double k1 = det(b - a, c - a), k2 = det(b - a, d - a);
    if (zero(k1 - k2)) return false;
    res = (d * k1 - c * k2) / (k1 - k2);
    return true;
}

void addedge(int x, int y) {
    e[tote] = y; prev[tote] = info[x]; info[x] = tote++;
    e[tote] = x; prev[tote] = info[y]; info[y] = tote++;
}

vector<double> Divide() {
    P = 0;
    for (int i = 0; i < N; ++i) {
        for (int j = i + 1; j < N; ++j) {
            if (intersect(a[i], b[i], a[j], b[j], p[P])) P++;
        }
    }
    sort(p, p + P);
    int tot = 1;
    for (int i = 1; i < P; ++i) if (!(p[i] == p[tot - 1]))
        p[tot++] = p[i];
    P = tot;
    memset(info, 0, sizeof info);
```

```
    tote = 2;
    for (int i = 0; i < N; ++i) {
        int last = -1;
        for (int j = 0; j < P; ++j)if(zero(det(b[i] - a[i], p[j] - a[i]))){
            if (last != -1) addedge(last, j);
            last = j;
        }
    }
    memset(mark, 0, sizeof mark);
    vector<double> area;
    for (int i = 2; i < tote; ++i) if (!mark[i]) {
        int laste = i ^ 1;
        int lastp = e[i];
        int head = e[laste];
        mark[i] = true;
        double ans;
        for (ans = det(p[head], p[lastp]); lastp != head; ) {
            double best = 1E20;
            int cur = -1;
            double base = getAngle(p[e[laste]] - p[lastp]);
            for (int k = info[lastp]; k; k = prev[k]) if (k != laste){
                double tmp = getAngle(p[e[k]] - p[lastp]) - base;
                if (tmp < 0) tmp += pi * 2;
                if (tmp >= pi * 2) tmp -= pi * 2;
                if (tmp < best) {
                    best = tmp;
                    cur = k;
                }
            }
            ans += det(p[lastp], p[e[cur]]);
            lastp = e[cur];
            laste = cur ^ 1;
            mark[cur] = true;
        }
        area.PB(fabs(ans) * .5);
    }
    sort(area.begin(), area.end());
    if (SIZE(area)) area.erase(area.end() - 1);
    return area;
}
```

【注释】

上述代码的复杂度为$O(N^4)$，如果提前对每个点上的边排序的话，可以快速找到下一条边，把其中的一个N转化成$\log N$。

【使用范例】

参见程序 SGU209.CPP。

第 4 章　数 据 结 构

4.1　二　叉　堆

【任务】

实现一个堆，实现插入、寻找最小值、修改任意元素、删除任意元素。

【说明】

由于堆是完全二叉树，我们使用下标从1开始的数组来表示这棵树，1代表根节点，对于每个节点i，它的左儿子为$i \times 2$，右儿子为$i \times 2 + 1$，父亲为$i/2$。

我们使用数组$heap$[]来记录堆中的元素。为了实现修改和删除操作我们额外使用id[]记录堆中位置为i的元素是第几个插入的，pos[]记录第i个插入的元素在堆中的位置。

实现堆核心函数为up(i)和down(i)，up(i)将堆中的位置为i的节点不断"上浮"（与父亲节点比较，如果小于父亲节点则与父亲节点交换），down(i)将堆中位置为i的节点不断"下沉"（与两个儿子节点比较，如果大于较小的儿子节点则与之交换）。

在插入一个值$value$时，我们将它加入堆的最底层（即$heap[++size] = value$），然后将其上浮；删除堆顶元素时，我们将堆顶与最后一个元素交换，然后下沉；修改元素时我们先利用pos数组找到它当前在堆中的位置，然后直接修改并调用up( )及down( )维护堆的性质即可；删除元素时我们将它修改为负无穷大，上浮到根，最后删除堆顶即可。

【接口】

结构体：$BinaryHeap$

成员变量：

int n	堆中当前元素个数
int $counter$	加入堆中的元素个数
int $heap$[ ]	堆中的元素
int id[ ]	队中位置为i的元素是第几个插入堆中的
int pos[ ]	第i个插入堆中的元素在堆中的位置

成员函数：

BinaryHeap( );	构造出的一个空堆
BinaryHeap(int array[ ], int offset);	将数组中的元素按顺序插入所构造的堆
复杂度：$O(n)$	
输　入：*array*[ ]	创建堆的元素所在的数组
offset	数组中需要用作创建堆的元素个数
void push(int v);	插入键值v
复杂度：$O(\log n)$	
int pop( );	删除堆顶元素
复杂度：$O(\log n)$	
输　出：堆顶元素插入堆中的次序编号	
int get(int i);	获取第i个插入堆中的元素值
复杂度：$O(1)$	
void change(int i, int value);	修改第i个元素为$value$
复杂度：$O(\log n)$	
void erase(int i);	删除第i个元素
复杂度：$O(\log n)$	

【代码】

```
const int MAXSIZE = 100000;      //二叉堆的大小
struct BinaryHeap {
    int heap[MAXSIZE], id[MAXSIZE], pos[MAXSIZE], n, counter;

    BinaryHeap() : n(0), counter(0) {}
    BinaryHeap(int array[], int offset) : n(0), counter(0) {
        for (int i = 0; i < offset; ++i) {
            heap[++n] = array[i];
            id[n] = pos[n] = n;
        }
        for (int i = n/2; i >= 1; --i) {
            down(i);
        }
    }

    void push(int v) {
        heap[++n] = v;
        id[n] = ++counter;
```

```
        pos[id[n]] = n;
        up(n);
    }

    int top() {
        return heap[1];
    }

    int pop() {
        swap(heap[1], heap[n]);
        swap(id[1], id[n--]);
        pos[id[1]] = 1;
        down(1);
        return id[n+1];
    }

    int get(int i) {
        return heap[pos[i]];
    }

    void change(int i, int value) {
        heap[pos[i]] = value;
        down(pos[i]);
        up(pos[i]);
    }

    void erase(int i) {
        heap[pos[i]] = INT_MIN;
        up(pos[i]);
        pop();
    }

    void up(int i) {
        int x = heap[i], y = id[i];

        for (int j = i/2; j >= 1; j /= 2) {
            if (heap[j] > x) {
                heap[i] = heap[j];
                id[i] = id[j];
```

```
                pos[id[i]] = i;
                i = j;
            } else {
                break;
            }
        }

        heap[i] = x;
        id[i] = y;
        pos[y] = i;
    }

    void down(int i) {
        int x = heap[i], y = id[i];

        for (int j = i*2; j <= n; j *= 2) {
            j += j < n && heap[j] > heap[j + 1];
        if (heap[j] < x) {
                heap[i] = heap[j];
                id[i] = id[j];
                pos[id[i]] = i;
                i = j;
            } else {
                break;
            }
        }

        heap[i] = x;
        id[i] = y;
        pos[y] = i;
    }

    bool empty() {
        return n == 0;
    }

    int size() {
        return n;
    }
```

```
};
```

【使用范例】

参见程序 POJ3268.CPP。

4.2 并 查 集

【任务】

维护一些不相交的集合，支持两种操作：合并两个集合，查询一个元素所处的集合。

【说明】

维护一个森林，每一棵树代表一个集合，树根元素为这个集合的代表元。利用数组$father$[]记录每个元素的父亲节点。

查询一个元素所处的集合时，只需不断寻找父亲节点，即可找到该元素所处集合的代表元。

合并两个集合时，先找到两个集合的代表元x, y，然后令$father[x] = y$即可。

优化1：路径压缩，即在沿着树根的路径找到元素a所在集合的代表元b之后，对这条路径上所有的元素x（包括a），直接令$father[x] = b$。

优化2：按$rank$启发式合并，即对于每个集合维护一个$rank$值，每次将$rank$较小的集合合并到$rank$较大的集合，合并两个$rank$相同的集合时$rank = rank + 1$。

【接口】

结构体：*DisjointSet*

成员变量：

vector<int> $father$　元素的父亲节点，树根元素的父亲为自身

vector<int> $rank$　树根元素所代表集合的$rank$

成员函数：

DisjointSet(int n);　初始化，n个元素，处于单独集合

复杂度：$O(n)$

int find(int v);　查找v所在集合的代表元

复杂度：均摊$O(1)$

void merge(int x, int y);　合并x所在集合与y所在集合

复杂度：均摊$O(1)$

【代码】

```
struct DisjointSet {
```

```
    std::vector<int> father, rank;

    DisjointSet(int n) : father(n), rank(n) {
        for (int i = 0; i < n; ++i) {
            father[i] = i;
        }
    }

    int find(int v) {
        return father[v] = father[v]==v ? v : find(father[v]);
    }

    void merge(int x, int y) {
        int a = find(x), b = find(y);
        if (rank[a] < rank[b]) {
            father[a] = b;
        } else {
            father[b] = a;
            if (rank[b] == rank[a]) {
                ++rank[a];
            }
        }
    }
};
```

【使用范例】

参见程序 POJ2492.CPP。

4.3 树状数组

【任务】

对于数组$A[1..n]$，在$O(\log n)$的时间内完成以下任务：

（1）给$A[i]$加上一个数

（2）求$A[1]+\cdots+A[i]$的和

【说明】

树状数组的第i个元素$Tree[i]$表示$A[\text{lowbit}(i)+1..i]$的和，其中$\text{lowbit}(i)$表示i的最低二进制位。

当想要查询一个$A[1]+\cdots+A[i]$的和，可以依据如下算法即可：

（1）令$sum=0$，转第（2）步。

（2）假如$i\leqslant 0$，算法结束，返回sum值，否则$sum+=Tree[i]$，转第（3）步。

（3）$i-=\text{lowbit}(i)$，转第（2）步。

可以看出，这个算法就是将这一个个区间的和全部加起来，为什么效率是$O(\log n)$的呢？以下给出证明：

$i-=\text{lowbit}(i)$这一步实际上等价于将i的二进制的最后一个1减去。而i的二进制里最多有$\log n$个1，所以查询效率是$O(\log n)$的。

而给$A[i]$加上x的算法如下：

（1）当$i>n$时，算法结束，否则转第（2）步。

（2）$Tree[i]+=x, i+=\text{lowbit}(i)$，转第（1）步。

$i+=\text{lowbit}(i)$这个过程实际上也只是一个把末尾1补为0的过程。容易看出复杂度也是$O(\log n)$的。

最后，lowbit(i)的求法有个简单的公式，$\text{lowbit}(i)=i\&(-i)$。

【接口】

void add(int x,int value);

复杂度：$O(\log n)$

输　入：$x, value$　$A[x]$增加$value$

int get(int x);

复杂度：$O(\log n)$

输　入：x　查询$A[1]$~$A[x]$的和

输　出：$A[1]+\cdots+A[x]$

【代码】

```
//maxn 为最大容量
const int maxn=100000;
int Tree[maxn+10];
inline int lowbit(int x)
{
    return(x&-x);
}
void add(int x,int value)
{
    for (int i=x;i<=maxn;i+=lowbit(i))
        Tree[i]+=value;
```

```
}
int get(int x)
{
    int sum=0;
    for (int i=x;i;i-=lowbit(i))
        sum+=Tree[i];
    return (sum);
}
```

【使用范例】

参见程序 POJ2352.CPP。

4.4 左 偏 树

【任务】

要求实现一个最小优先队列，使得插入、删除、合并等操作均在$O(\log n)$的时间复杂度以内完成。

【说明】

左偏树是一个堆，为了实现快速合并的操作，我们可以构造一棵二叉树，使得并且右子树尽量简短。这里我们可以定义一个左偏树的外节点，是一个左子树为空或者右子树为空的节点，对于每个点定义一个距离$dist$为它到它子树内外节点的最短距离。一个合法的左偏树节点需要满足堆性以及它的右子树的$dist$比左子树的$dist$小，这样右子树的$dist$是严格在$\log n$以内的。于是我们在合并的时候，将另一个左偏树与当前左偏树的右子树合并，这样递归下去，则时间复杂度是$O(\log n)$的。

【接口】

int Init(int x);

输入：x　单节点左偏树的权值

输出：新建的左偏树的编号

int Insert(int x, int y);

复杂度：$O(\log n)$

输　入：x,y　向编号为x的左偏树中插入一个权值为y的节点

输　出：新的堆顶编号

int Top(int x);

复杂度：$O(1)$

输　入：x　左偏树的编号
输　出：编号为x的左偏树的堆顶的权值

int Pop(int x);
复杂度：$O(\log n)$
输　入：x　左偏树的编号
输　出：删除编号为x的左偏树的堆顶，返回新的堆顶编号

int Merge(int x,int y);
复杂度：$O(\log n)$
输　入：x,y　要合并的两棵左边树的编号
输　出：新的堆顶编号

【代码】

```
//tot 为添加过的节点个数,maxn 为最多节点数
const int maxn=100000;
int tot,v[maxn],l[maxn],r[maxn],d[maxn];
int Merge(int x,int y)
{
    if (!x)
        return (y);
    if (!y)
        return (x);
    if (v[x]<v[y])
        swap(x,y);
    r[x]=Merge(r[x],y);
    if (d[l[x]]<d[r[x]])
        swap(l[x],r[x]);
    d[x]=d[r[x]]+1;
    return (x);
}
int Init(int x)
{
    tot++;
    v[tot]=x;
    l[tot]=r[tot]=d[tot]=0;
}
int Insert(int x,int y)
{
    return (Merge(x,Init(y)));
```

```
}
int Top(int x)
{
    return (v[x]);
}
int Pop(int x)
{
    return (Merge(l[x],r[x]));
}
```

【注释】

以上的Insert，Top，Pop，Merge操作都要求以左偏树堆顶的编号代表这一棵左偏树。

【使用范例】

参见程序 ZOJ2334.CPP。

4.5 Trie

【任务】

设计一种数据结构，支持两种操作：插入一个字符串；查询一个字符串是否存在。

【说明】

我们采用2个数组实现一个Trie树。$child[i][j]$代表以i为根的子树，字符j代表的边连向哪一个节点（如果$child[i][j]=0$，则说明没有对应的节点）。初始时根节点为1。$flag[i]$代表节点i是否为一个单词的结尾。

插入时，我们从根节点沿着字符串的每个字符走向下一层节点。如果该节点不存在则分配一个新节点。对于最后插入的节点i，我们令$flag[i]=\text{true}$。

查找和插入的过程基本相同，区别是：如果我们走到一个不存在的节点那么返回查找失败。如果最后我们停留在某个节点i，那么返回$flag[i]$即可。

【接口】

结构体：*Trie*

成员函数：

void insert(const char *str);　　插入字符串*str*

复杂度：$O(Length)$

bool query(const char *str);　　查询字符串*str*是否出现

复杂度：$O(Length)$

【代码】

```
//CHARSET 为字符集大小
//BASE 为字符集 ASCII 最小字符
//MAX_NODE 为最大点数
const int CHARSET=26,BASE='a',MAX_NODE=100000;
struct Trie
{
    int tot,root,child[MAX_NODE][CHARSET];
    bool flag[MAX_NODE];
    Trie()
    {
        memset(child[1],0,sizeof(child[1]));
        flag[1]=false;
        root=tot=1;
    }
    void insert(const char *str)
    {
        int *cur=&root;
        for (const char *p=str;*p;++p)
        {
            cur=&child[*cur][*p-BASE];
            if (*cur==0)
            {
                *cur=++tot;
                memset(child[tot],0,sizeof(child[tot]));
                flag[tot]=false;
            }
        }
        flag[*cur]=true;
    }
    bool query(const char *str)
    {
        int *cur=&root;
        for (const char *p=str;*p && *cur;++p)
            cur=&child[*cur][*p-BASE];
        return(*cur && flag[*cur]);
    }
};
```

【使用范例】

参见程序 POJ1056.CPP。

4.6 Treap

【任务】

要求动态维护一个有序表，支持在$O(\log N)$的时间内完成插入一个元素，删除一个元素和查找第K大元素的任务。

【说明】

Treap是一种平衡化二叉搜索树，在键值$key[]$满足二叉搜索树要求的前提下，增加了$priority[\,]$是满足堆序的条件。可以证明，如果$priority[\,]$是随机的，那么Treap的期望深度是$O(\log N)$的，也就是说，大部分操作可以在$O(\log N)$的时间内完成。

在Treap插入节点时，先把节点插入到二叉树对应的位置，并随机给定一个权值，然后类似堆一样，每次比较该节点与其父亲，如果不满足堆的要求，那么交换两个节点，类似于堆，只需$O(\log N)$次就可完成性质的维护。

在Treap中删除节点非常简单，先把节点的$priority[]$修改成充分大，然后进行堆序的维护，在叶子处直接删除即可。

程序实现时，用$childs[0]$和$childs[1]$分别表示左右儿子，这样可以把两种旋转统一起来，简化代码。

【接口】

结构体：*Treap*

成员函数：

void insert(int k);

复杂度：$O(\log N)$

输　入：x　插入元素的值

void erase(int k);

复杂度：$O(\log N)$

输　入：k　删除元素的值

int getKth(int k);

复杂度：$O(\log N)$

输　入：k　要查找第k大元素

输　出：有序表中的第k大元素

【代码】

```
const int maxNode=444444;

struct Treap{
    int root,treapCnt,key[maxNode],priority[maxNode],
    childs [maxNode][2],cnt[maxNode],size[maxNode];

    Treap(){
        root=0;
        treapCnt=1;
        priority[0]=INT_MAX;
        size[0]=0;
    }

    void update(int x){
        size[x]=size[childs[x][0]]+cnt[x]+size[childs[x][1]];
    }

    void rotate(int &x,int t){
        int y=childs[x][t];
        childs[x][t]=childs[y][1-t];
        childs[y][1-t]=x;
        update(x);
        update(y);
        x=y;
    }

    void __insert(int &x,int k){
        if(x){
            if(key[x]==k){
                cnt[x]++;
            }else{
                int t=key[x]<k;
                __insert(childs[x][t],k);
                if(priority[childs[x][t]]<priority[x]){
                    rotate(x,t);
                }
            }
```

```
        }else{
            x=treapCnt++;
            key[x]=k;
            cnt[x]=1;
            priority[x]=rand();
            childs[x][0]=childs[x][1]=0;
        }
        update(x);
    }

    void __erase(int &x,int k){
        if(key[x]==k){
            if(cnt[x]>1){
                cnt[x]--;
            }else{
                if(childs[x][0]==0&&childs[x][1]==0){
                    x=0;
                    return;
                }
                intt=priority[childs[x][0]]>priority[childs[x][1]];
                rotate(x,t);
                __erase(x,k);
            }
        }else{
            __erase(childs[x][key[x]<k],k);
        }
        update(x);
    }

    int __getKth(int &x,int k){
        if(k<=size[childs[x][0]]){
            return __getKth(childs[x][0],k);
        }
        k-=size[childs[x][0]]+cnt[x];
        if(k<=0){
            return key[x];
        }
        return __getKth(childs[x][1],k);
    }
```

```
    void insert(int k){
        __insert(root,k);
    }

    void erase(int k){
        __erase(root,k);
    }

    int getKth(int k){
        return __getKth(root,k);
    }
};
```

【使用范例】

参见程序 POJ2985.CPP。

4.7 伸 展 树

【任务】

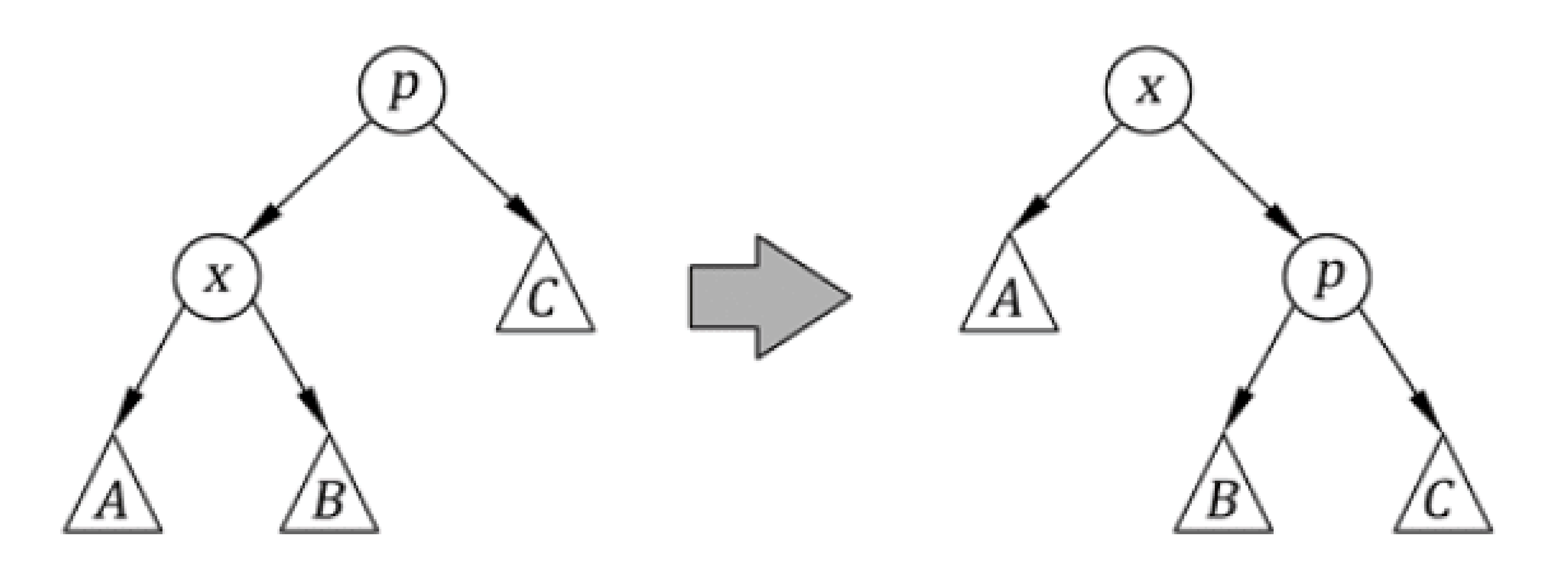
要求实现能够在O(log$N$)时间复杂度内实现各类二叉查找树操作的数据结构。

【说明】

Splay的本质是一棵平衡二叉查找树。普通的二叉查找树在某些情况下会退化成一条链的情况，一般的平衡二叉查找树都是利用旋转操作来保证一些性质使得整棵树的平衡，而Splay这里采用的是splay（伸展）操作来使得在均摊情况下时间复杂度是O(log$N$)的。

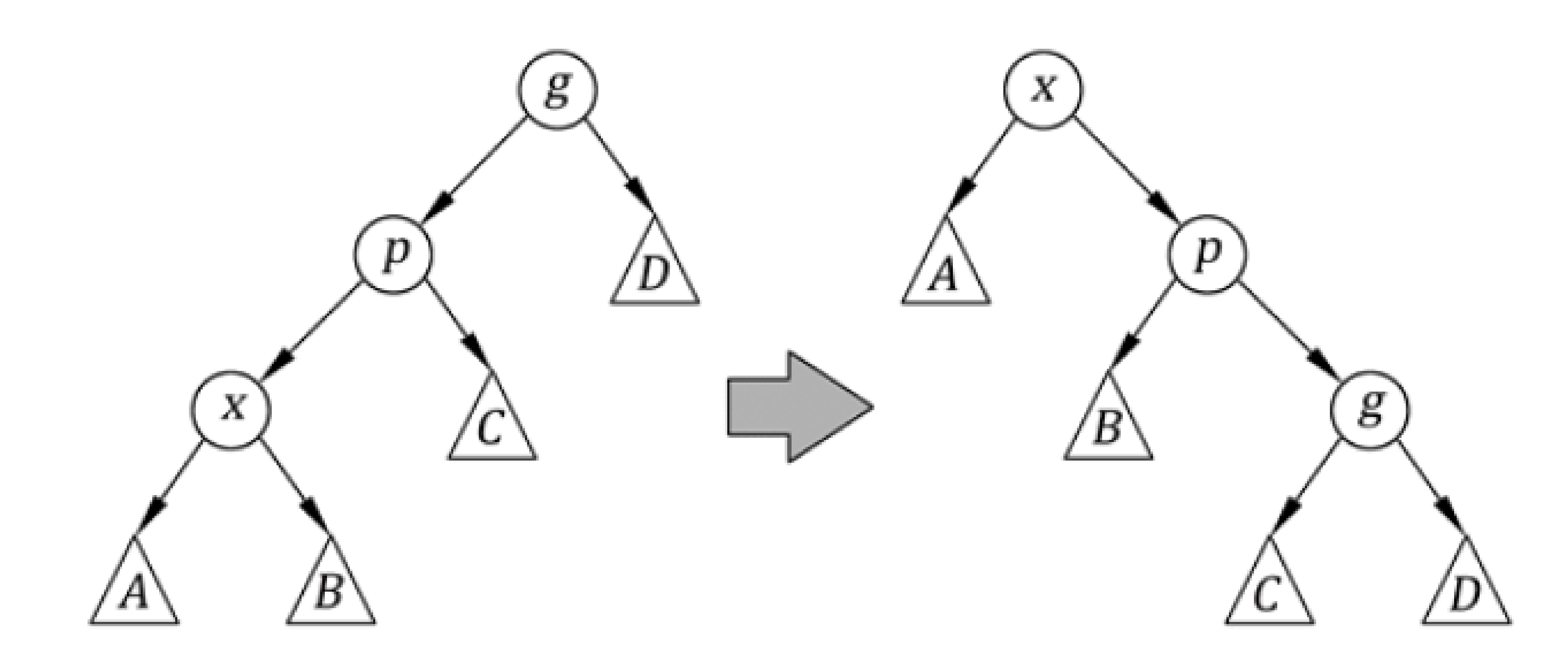
splay操作依靠两种旋转：Zig（左旋）和Zag（右旋）以及它们衍生出来的四种双旋来实现把一个节点旋转到根的操作。并可以证明splay操作的均摊复杂度是O(log$N$)的。

Zig操作：当x的父亲p是原树的根，并且x是p的左儿子的时候进行，效果如图所示(Zag操作类似)。

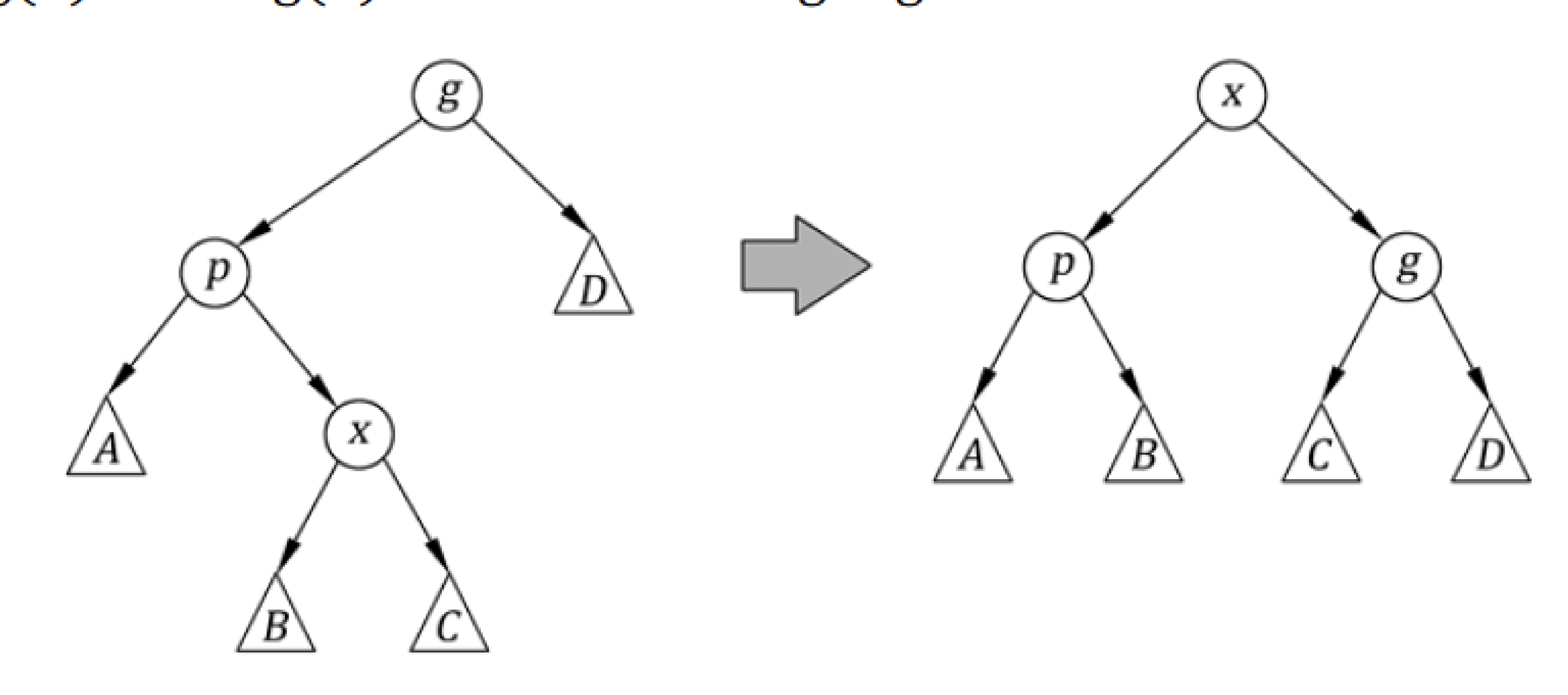


Zig-Zig操作：当x是父亲p的左儿子，p有父亲并且p是父亲g的左儿子的时候进行，效

果等价于先Zig(p)，再Zig(x)，如图所示（Zag-Zag操作类似）。



Zig-Zag操作：当x是父亲p的左儿子，p有父亲并且p是父亲g的右儿子的时候进行，效果等价于先Zig(x)，再Zag(x)，如图所示（Zag-Zig操作类似）。



插入操作：和普通二叉查找树类似地把一个节点插入，之后对该节点进行splay操作。

删除操作：可以像普通二叉查找树一样删除一个节点，再对它的父亲进行splay操作。

在一些附加信息以及标记传递的帮助下，Splay可以比线段树更灵活地维护一些区间信息：我们可以把Splay里的每个节点用来表示一个单位区间，而对区间$[a,b]$进行操作的时候，我们可以把$(a-1)$元素旋到树根，再把$(b+1)$元素旋到$(a-1)$元素的右子树的树根，这样$(b+1)$元素的左子树就代表了整个区间$[a,b]$，我们就可以在上面进行各类操作，例如区间翻转等等。

【接口】

结构体：*SplayTree*

成员函数：

void split(int &x, int &y, int a);

复杂度：均摊O(log$N$)

输　入：x　　分裂之前Splay树的树根

　　　　a　　分裂之后前半部分的大小

输　出：x,y分裂之后两棵树的树根

void join(int &x, int &y);
复杂度：均摊O(log$N$)
输　入：x, y　　合并之前两棵树的树根
输　出：x合并之后Splay树的树根

int getRank(int &x);
复杂度：均摊O(log$N$)
输　入：x　　Splay树的节点
输　出：节点在树中的排名

void split3(int &x, int &y, int &z, int a, int b);
复杂度：均摊O(log$N$)
输　入：a, b　　将树分成 3 部分，第一部分为前$a-1$个，第二部分为a至b，第三部分为b之后的
输　出：x, y, z分别为三部分的树根

void join3(int &x, int y, int z);
复杂度：均摊O(log$N$)
输　入：x, y, z　　三棵树的树根；将树x, y, z合并
输　出：x　　树的树根

void reverse(int a, int b);
将树从a至b的区间翻转

【代码】

```
struct SplayTree {
    int nodeCnt, root, type[maxNodeCnt], parent[maxNodeCnt],
    childs[maxNodeCnt][2], size[maxNodeCnt],
    stack[maxNodeCnt], reversed[maxNodeCnt];

    void clear() {
        root = 0;
        size[0] = 0;
        nodeCnt = 1;
    }

    int malloc() {
        type[nodeCnt] = 2;
        childs[nodeCnt][0] = childs[nodeCnt][1] = 0;
        size[nodeCnt] = 1;
```

```
        reversed[nodeCnt] = 0;
        return nodeCnt++;
    }

    void update(int x) {
        size[x] = size[childs[x][0]] + 1 + size[childs[x][1]];
    }

    void pass(int x) {
        // NOTICE: childs[x][i] == 0
        if (reversed[x]) {
            swap(childs[x][0], childs[x][1]);
            type[childs[x][0]] = 0;
            reversed[childs[x][0]] ^= 1;
            type[childs[x][1]] = 1;
            reversed[childs[x][1]] ^= 1;
            reversed[x] = 0;
        }
    }

    void rotate(int x) {
        int t = type[x],
            y = parent[x],
            z = childs[x][1 - t];
        type[x] = type[y];
        parent[x] = parent[y];
        if (type[x] != 2) {
            childs[parent[x]][type[x]] = x;
        }
        type[y] = 1 - t;
        parent[y] = x;
        childs[x][1 - t] = y;
        if (z) {
            type[z] = t;
            parent[z] = y;
        }
        childs[y][t] = z;
        update(y);
    }
```

```

void splay(int x) {
    int stackCnt = 0;
    stack[stackCnt++] = x;
    for (int i = x; type[i] != 2; i = parent[i]) {
        stack[stackCnt ++] = parent[i];
    }
    for (int i = stackCnt - 1; i > -1; --i) {
        pass(stack[i]);
    }
    while (type[x] != 2) {
        int y = parent[x];
        if (type[x] == type[y]) {
            rotate(y);
        } else {
            rotate(x);
        }
        if (type[x] == 2) {
            break;
        }
        rotate(x);
    }
    update(x);
}

int find(int x, int rank) {
    while (true) {
        pass(x);
        if (size[childs[x][0]] + 1 == rank) {
            break;
        }
        if (rank <= size[childs[x][0]]) {
            x = childs[x][0];
        } else {
            rank -= size[childs[x][0]] + 1;
            x = childs[x][1];
        }
    }
    return x;
```

```
    }

    void split(int &x, int &y, int a) {
        // NOTICE: x, y != 0
        y = find(x, a + 1);
        splay(y);
        x = childs[y][0];
        type[x] = 2;
        childs[y][0] = 0;
        update(y);
    }

    void split3(int &x, int &y, int &z, int a, int b) {
        split(x, z, b);
        split(x, y, a - 1);
    }

    void join(int &x, int y) {
        // NOTICE x, y != 0
        x = find(x, size[x]);
        splay(x);
        childs[x][1] = y;
        type[y] = 1;
        parent[y] = x;
        update(x);
    }

    void join3(int &x, int y, int z) {
        join(y, z);
        join(x, y);
    }

    int getRank(int x) {
        splay(x);
        root = x;
        return size[childs[x][0]];
    }

    void reverse(int a, int b) {
```

```
        int x, y;
        split3(root, x, y, a + 1, b + 1);
        reversed[x] ^= 1;
        join3(root, x, y);
    }
}
```

【使用范例】

参见程序 CERC2007I.CPP。

4.8　RMQ 线段树

【任务】

要求实现一种数据结构，使得它能够在$O(\log N)$时间复杂度内动态维护一段序列：修改一个元素的权值，询问一段区间内的最小（最大）值。

【说明】

线段树是一个二叉树，树上每个节点表示一段区间，每个节点的左右儿子分别是该节点表示的区间从中间断开后分成的左区间和右区间。每个区间记录一个当前子树的最小/最大值$Top[i]$。

查询$[a,b]$区间的话也很简单，对于当前节点i，如果$[a,b]$能够完全覆盖i表示的区间，则直接返回$Top[i]$，否则判断与左右区间是否有交递归进入访问，取两者之间的最大值即可。

【接口】

（假设这里维护的是最大值）

类：*IntervalTree*

成员函数：

IntervalTree(int size);　　构造一棵维护区间$[1..size]$的线段树

int Query(int a,int b);　　查询$[a..b]$区间内的最大值

复杂度：$O(\log N)$

void Modify(int a,int d);　　把第a个元素的值改成d

复杂度：$O(\log N)$

【代码】

```
#define TREE_SIZE (1<<(20))
class IntervalTree{
```

```
    private:
        int Cover[TREE_SIZE],Top[TREE_SIZE];
        int size;
        int _Query(int a,int b,int l,int r,int Ind){
            if(a<=l&&b>=r)returnTop[Ind];
            int mid=(l+r)>>1,ret=Cover[Ind];
            if(a<=mid)ret=max(ret,_Query(a,b,l,mid,Ind<<1));
            if(b>mid)ret=max(ret,_Query(a,b,mid+1,r,(Ind<<1)+1));
            return ret;
        }
        void _Modify(int a,int l,int r,int Ind,int d){
            if(l==r&&l==a){
                Cover[Ind]=Top[Ind]=d;
                return;
            }
            int mid=(l+r)>>1;
            if(a<=mid)_Modify(a,l,mid,Ind<<1,d);
            else _Modify(a,mid+1,r,(Ind<<1)+1,d);
            Top[Ind]=max(Top[Ind<<1],Top[(Ind<<1)+1]);
        }
    public:
        IntervalTree(){
            memset(Cover,0,sizeof(Cover));
            memset(Top,0,sizeof(Top));
            size=(TREE_SIZE>>2)-1;
        }
        IntervalTree(int size):size(size){
            memset(Cover,0,sizeof(Cover));
            memset(Top,0,sizeof(Top));
        }
        int Query(int a,int b){return _Query(a,b,1,size,1);}
        void Modify(int a,int d){
            return _Modify(a,1,size,1,d);
        }
};
```

【使用范例】

参见程序 POJ3264_2.CPP。

4.9　ST 表

【任务】

给定一个数组$A[n]$，动态查询数组元素$A[l], A[l+1], \cdots, A[r]$的最小值。

【说明】

我们首先使用$O(n\log n)$的时间预处理出数组$st[i][j]$，代表从$A[i]$开始连续2^j个元素中的最小值。可以使用动态规划，状态转移方程如下：

边界条件为$st[i][0] = A[i]$。

对于一个询问$[L, R]$，我们令$k = \text{floor}(\log_2(R - L + 1))$，那么$[L, R]$中的最小值就是$\min(st[L][k], st[R - 2^i + 1][k])$。

【接口】

void st_prepare(int n, int *array);

复杂度：$O(n\log n)$

输　入：n　　数组长度

　　　　$array$　　数组

int query_min(int l, int r);

复杂度：$O(1)$

输　入：l, r　　询问区间的两个端点

输　出：$A[l], A[l+1], \cdots, A[r]$的最小值

【代码】

```
const int MAX = 100000;
int stTable[MAX][32];
int preLog2[MAX];

void st_prepare(int n, int *array) {
    preLog2[1] = 0;
    for (int i = 2; i <= n; ++i) {
        preLog2[i] = preLog2[i-1];
        if ((1 << preLog2[i] + 1) == i) {
            ++preLog2[i];
        }
    }
    for (int i = n-1; i >= 0; --i) {
```

```
            stTable[i][0] = array[i];
            for (int j = 1; (i + (1 << j) - 1) < n; ++j) {
                stTable[i][j] = min(stTable[i][j - 1], stTable[i + (1
                                << j-1)][j - 1]);
            }
        }
    }
    int query_min(int l, int r) {
        int len = r - l + 1, k = preLog2[len];
        return min(stTable[l][k], stTable[r - (1<<k) + 1][k]);
    }
```

【使用范例】

参见程序 POJ3264.CPP。

4.10 动 态 树

【任务】

要求实现一种数据结构，使得它能够在$O(\log N)$时间复杂度内维护一个包含N个点的森林，并且支持形态和权值信息的操作，形态操作包括在两点直接连边以及删除一条边。

【说明】

可以想到，如果没有对树的形态的改变的话，我们可以把树剖分成许多条链并维护上面的权值信息。然而这次多了树的形态的改变，也就是说随着树的形态的改变，链的剖分方案也要跟着改变，于是我们可以借鉴Splay的思想来动态维护许多树链（通称Link-cut Tree）：

Link-cut Tree对于每个节点定义一个$preferred\ child$为最近一次访问的儿子（最近一次被访问的节点没有$preferred\ child$），$preferred\ edge$为每个节点到$preferred\ child$的边，$preferred\ path$为连续的$preferred\ edge$组成的路径。由于访问节点后$preferred\ path$会改变，所以每条$preferred\ path$必须要用Splay维护区间的权值信息。

于是我们有了访问操作Expose(x)，他可以找到x到树根的一条路径（最后记住要对x节点进行splay操作）。对于两点(a,b)连边，假设连边后b是a的儿子，我们可以先Expose(a)以及Expose(b)，然后把b的路径上全部翻转一次（可以利用Splay的标记传递实现），接着把b接到a的右儿子中（Splay中）。对于删除一条边的操作，假设删除(a,b)这条边，b是a的儿子，就Expose(b)，然后把b在Splay中与左儿子（就是a）的连边删除就可以了。

可以证明以上的均摊时间复杂度是$O(\log N)$的。

【接口】

（这里假定维护的是树上两点间边权的和）

int Expose(int u);

复杂度：均摊O(log N)

输　入：u　　　进行访问操作的节点

输　出：u所在树的树根

int Query(int x,int y);

复杂度：均摊O(log N)

输　入：x,y　　询问的两个节点

输　出：若x,y在一棵树里，返回他们在树上的路径的权值和，否则返回-1

void Join(int x,int y);

复杂度：均摊O(log N)

输　入：x,y　　从x往y添一条边（把x作为y的儿子）

void Cut(int x);

复杂度：均摊O(log N)

输　入：x　　　把x与根的连边删除

【代码】

```
int Lch[MaxNode];
int Rch[MaxNode];
int Pnt[MaxNode];
int Data[MaxNode];
int Sum[MaxNode];
int Rev[MaxNode];
int List[MaxNode];
int Total;

inline bool isRoot(int t){
    return(!Pnt[t]||(Lch[Pnt[t]]!=t&&Rch[Pnt[t]]!=t));
}
inline void swap(int &a,int &b){
    int c=a;a=b;b=c;
}
void Reverse(int cur){
    if(!Rev[cur])
        return;
```

```
    swap(Lch[cur],Rch[cur]);
    Rev[Lch[cur]]^=1;
    Rev[Rch[cur]]^=1;
    Rev[cur]=0;
}
inline void Update(int cur){
    Sum[cur]=Sum[Lch[cur]]+Sum[Rch[cur]]+Data[cur];
}
void LeftRotate(int cur){
    if(isRoot(cur))
        return;
    int pnt=Pnt[cur],anc=Pnt[pnt];
    Lch[pnt]=Rch[cur];
    if(Rch[cur])
        Pnt[Rch[cur]]=pnt;
    Rch[cur]=pnt;
    Pnt[pnt]=cur;
    Pnt[cur]=anc;
    if(anc){
        if(Lch[anc]==pnt)
            Lch[anc]=cur;
        else if(Rch[anc]==pnt)
            Rch[anc]=cur;
    }
    Update(pnt);
    Update(cur);
}
void RightRotate(int cur){
    if(isRoot(cur))
        return;
    int pnt=Pnt[cur],anc=Pnt[pnt];
    Rch[pnt]=Lch[cur];
    if(Lch[cur])
        Pnt[Lch[cur]]=pnt;
    Lch[cur]=pnt;
    Pnt[pnt]=cur;
    Pnt[cur]=anc;
    if(anc){
        if(Lch[anc]==pnt)
```

```
                Lch[anc]=cur;
            else if(Rch[anc]==pnt)
                Rch[anc]=cur;
        }
        Update(pnt);
        Update(cur);
}
void Splay(int cur){
        int pnt,anc;
        List[++Total]=cur;
        for(int i=cur;!isRoot(i);i=Pnt[i])
            List[++Total]=Pnt[i];
        for(;Total;--Total)
            if(Rev[List[Total]])
                Reverse(List[Total]);
        while(!isRoot(cur)){
            pnt=Pnt[cur];
            if(isRoot(pnt)){
                if(Lch[pnt]==cur)
                    LeftRotate(cur);
                else
                    RightRotate(cur);
            }else{
                anc=Pnt[pnt];
                if(Lch[anc]==pnt){
                    if(Lch[pnt]==cur)
                        LeftRotate(pnt),LeftRotate(cur);
                    else
                        RightRotate(cur),LeftRotate(cur);
                }else{
                    if(Lch[pnt]==cur)
                        LeftRotate(cur),RightRotate(cur);
                    else
                        RightRotate(pnt),RightRotate(cur);
                }
            }
        }
}
int Expose(int u){
```

```
    int v=0;
    for(;u;u=Pnt[u])
        Splay(u),Rch[u]=v,v=u,Update(u);
    for(;Lch[v];v=Lch[v]);
    return v;
}
void Modify(int x,int d){
    Splay(x);
    Data[x]=d;
    Update(x);
}
int Query(int x,int y){
    int rx=Expose(x),ry=Expose(y);
    if(rx!=ry)
        return -1;
    else{
        for(int u=x,v=0;u;u=Pnt[u]){
            Splay(u);
            if(!Pnt[u])
                return Sum[Rch[u]]+Data[u]+Sum[v];
            Rch[u]=v;
            Update(u);
            v=u;
        }
    }
}
void Join(int x,int y){
    int rx=Expose(x),ry=Expose(y);
    if(rx==ry)
        puts("no");
    else{
        puts("yes");
        Splay(x);
        Rch[x]=0;
        Rev[x]=1;
        Pnt[x]=y;
        Update(x);
    }
}
```

```

void Cut(int x){
    if(Pnt[x]){
        int rx=Expose(x);
        Pnt[Lch[x]]=0;
        Lch[x]=0;
        Update(x);
    }
}
```

【使用范例】

参见程序 SPOJ_OTOCI.CPP。

4.11　块状链表

【任务】

设计一种数据结构在$O(\sqrt{n})$的时间内完成对一个有序表的插入、删除、查询等操作。

【说明】

我们考虑一个链表，它的每个节点存储着一个数组，以这个来存储一个有序表。我们称每个链表的节点为一个块。假设有序表的规模为N，链表的规模为n，每个块的数组的规模为m，显然有$N = n \times m$。考虑插入、删除、查询等操作，都是首先遍历每块，然后在某块中进行操作，所以复杂度均为$O(n + m)$，那么我们让n与m都为$\sqrt{N}$级别的话，就可以让每个操作的复杂度变为$O(\sqrt{N})$。

对于插入操作，首先找到应该插入的块，然后移动数组插入，如果这块的规模已经达到了$2\sqrt{N}$，我们应该把这块拆成两块，以保证时间复杂度。删除与查询操作不会使元素变多，所以不会提高时间复杂度。这样的算法可以保证m的规模在$\sqrt{N}$与$2\sqrt{N}$之间，那么n的规模不会超过$\sqrt{N}$，那么复杂度便可保证在$O(\sqrt{N})$的级别了。

【接口】

void insert(int x, int pos);

复杂度：$O(\sqrt{N})$

输　入：x　　插入的值

　　　　pos　　插入的位置

void del(int pos);

复杂度：$O(\sqrt{N})$

输　入：*pos*　　删除的数的位置

int find(int pos);

复杂度：$O(\sqrt{N})$

输　入：*pos*　　查询的数的位置

输　出：第*pos*位置的数

【代码】

```
//m 为 sqrt(N)级别的一个数
const int m=350;
struct data
{
    int s,a[2*m+5];
    data *next;
};
data *root;
void insert(int x,int pos)
{
    if (root==NULL)
    {
        root=new(data);
        root->s=1;
        root->a[1]=x;
        return;
    }
    data *k=root;
    while (pos>k->s && k->next!=NULL)
    {
        pos-=k->s;
        k=k->next;
    }
    memmove(k->a+pos+1,k->a+pos,sizeof(int)*(k->s-pos+1));
    k->s++;
    k->a[pos]=x;
    if (k->s==2*m)
    {
        data *t=new(data);
        t->next=k->next;
        k->next=t;
        memcpy(t->a+1,k->a+m+1,sizeof(int)*m);
```

```
        t->s=k->s=m;
    }
}
void del(int pos)
{
    data *k=root;
    while (pos>k->s && k->next!=NULL)
    {
        pos-=k->s;
        k=k->next;
    }
    memmove(k->a+pos,k->a+pos+1,sizeof(int)*(k->s-pos));
    k->s--;
}
int find(int pos)
{
    data *k=root;
    while (pos>k->s && k->next!=NULL)
    {
        pos-=k->s;
        k=k->next;
    }
    return(k->a[pos]);
}
void destroy(data *k)
{
    if (k->next!=NULL)
        destroy(k->next);
    delete(k);
}
```

【注释】

在操作之前，请初始化头指针：$root$ = NULL。

在操作完成后，执行destroy($root$)释放空间。

请注意，对于insert，del，find操作，请保证参数pos的合法性。

【使用范例】

参见程序 SPOJ_QMAX3VN.CPP。

4.12 树链剖分

【任务】

给定一棵树，将它划分成若干条互不相交的路径，满足：从节点$u \rightarrow v$最多经过$\log N$条路径以及$\log N$条不在路径上的边。

【说明】

我们使用以下几个数组来描述剖分出来的路径：

$Belong[v]$	节点v所属的路径编号
$Idx[v]$	节点v在其路径中的编号，节点按深度由深到浅依次标号
$Head[p]$	编号为p的路径的顶端节点
$Len[p]$	路径p的长度
$Dep[v]$	节点v的深度
$Father[v]$	节点v的父亲节点
$Size[v]$	以节点v为根的子树的节点个数

划分操作采用BFS以避免栈空间溢出。按照BFS的发现顺序逆序处理，对于每一个节点v，找到它的$size$最大的子节点u。如果u不存在，那么给v分配一条新的路径，否则v就延续u所属的路径。

查询两个节点u、v之间的路径时，首先判断它们是否属于同一条路径。如果是则直接在这条路径上查询并返回，否则选择所属路径顶端节点h的深度较大的节点（不妨设是v），查询v到h，并令 $v = father[h]$继续查询，直到u、v属于同一条路径。

【接口】

void insert(int x, int y);

输　入：x, y　　添加一条x到y的边

void split( );

复杂度：$O(n\log n)$

输　入：	$Prev$	$Prev[i]$表示邻接表中，第i条边在链表中的下一条边
	$info$	$info[v]$表示邻接表中，从点v出发的边链表的头节点
输　出：	$Belong$	$Belong[v]$表示节点v所属的路径编号
	Idx	$Idx[v]$表示节点v在其路径中的编号 按深度由深到浅依次标号
	$Head$	$Head[p]$表示编号为p的路径的顶端节点
	Len	$Len[p]$表示路径p的长度

Dep	*Dep*[*v*]表示节点*v*的深度
Father	*Father*[*v*]表示节点*v*的父亲节点
Size	*Size*[*v*]表示以节点*v*为根的子树的节点个数

【代码】

```
const int maxn = 100000 + 5;
const int maxm = maxn + maxn;
int v[maxm];
int Prev[maxm];
int info[maxn];
int Q[maxn];
int idx[maxn];
int dep[maxn];
int size[maxn];
int belong[maxn];
int father[maxn];
bool vis[maxn];
int head[maxn];
int len[maxn];
int l, r, ans, cnt=0;
int N, nedge = 0;

inline void insert(int x, int y) {
    ++nedge;
    v[nedge] = y; Prev[nedge] = info[x]; info[x] = nedge;
}

void split() {
    memset(dep, -1, sizeof(dep));
    l = 0;
    dep[ Q[ r = 1 ] = 1 ] = 0;
    father[1] = -1;
    while(l < r) {
        int x = Q[++l];
        vis[x] = false;
        for(int y = info[x]; y ; y = Prev[y])
            if(dep[v[y]] == -1) {
                dep[ Q[++ r] = v[y] ] = dep[x] + 1;
                father[v[y]] = x;
```

```
            }
    }
    for(int i = N; i ; i--) {
        int x = Q[i], p = -1;
        size[x] = 1;
        for(int y = info[x]; y; y = Prev[y])
            if(vis[v[y]]) {
                size[x] += size[v[y]];
                if(p == -1 || size[v[y]] > size[p])
                    p = v[y];
            }
        if(p == -1) {
            idx[x] = len[++ cnt] = 1;
            belong[head[cnt] = x] = cnt;
        }
        else {
            idx[x] = ++ len[ belong[x] = belong[p] ];
            head[belong[x]] = x;
        }
        vis[x] = true;
    }
}
```

【注释】

用insert函数建图，然后调用split就可以完成剖分。点从1开始编号，并假设根节点是1。

【使用范例】

参见程序 URAL1553.CPP。

第 5 章 论题选编

5.1 字符串

5.1.1 KMP

【任务】

要求实现一种算法使得能够在线性复杂度内求出一个串在另一个串的所有匹配位置。

【说明】

设模板串是 $pattern$，令 $\text{next}[i] = \max\{k \mid pattern[0..k-1] = pattern[i-k+1..i]\}$，求解 $next[]$ 可以使用动态规划，即 $next[i+1]$ 可以由 $next[i]$, $next[next[i]]$, …得到。

得到 $next[]$ 数组之后，设两个指针 i 和 j，分别指向文本串和模式串，成功匹配得向后移动 j，否则把 j 移动到 $next[j]$。当 j 移动到模式串末尾时，就说明匹配成功。

【接口】

vector <int> find_substring(string pattern, string text);

复杂度：$O(N+M)$

输　入：$pattern$ 模式串

　　　　$text$ 　文本串

输　出：所有匹配点的下标

【代码】

```
vector <int> find_substring(string pattern, string text) {
    int n = pattern.size();
    vector <int> next(n + 1, 0);
    for (int i = 1; i < n; ++ i) {
        int j = i;
        while (j > 0) {
            j = next[j];
            if (pattern[j] == pattern[i]) {
```

```
                    next[i + 1] = j + 1;
                    break;
                }
            }
        }
        vector <int> positions;
        int m = text.size();
        for (int i = 0, j = 0; i < m; ++i) {
            if (j < n && text[i] == pattern[j]) {
                j++;
            } else {
                while (j > 0) {
                    j = next[j];
                    if (text[i] == pattern[j]) {
                        j++;
                        break;
                    }
                }
            }
            if (j == n) {
                positions.push_back(i - n + 1);
            }
        }
        return positions;
    }
```

【使用范例】

参见程序 KMP.CPP。

5.1.2 扩展 KMP

【任务】

要求实现一种算法使得能够在线性复杂度内求出一个串对于另一个串的每个后缀的最长公共前缀。

【说明】

假设两个串为s和p，要求p与每个s的后缀的最长公共前缀，我们可以先求出p与它自己的每个后缀的最长公共前缀（假设为A）。类似KMP算法的思想，需要利用好已知的信息，

假设我们现在要计算p的第i个字符开头的后缀，而我们已经得到了$A[1..i-1]$，我们可以找到以前的一个k，使得$k+A[k]-1$最大（就是被匹配到的范围最大），我们可以得知：$p[1..A[k]]=p[k..k+A[k]-1]$，于是可以得到$p[i..k+A[k]-1]=p[i-k+1..A[k]]$，即我们可以利用到$A[i-k+1]$的信息，分两种情况讨论，如果$i+A[i-k+1]-1$比$k+A[k]-1$小，则$A[i]$的值直接就是$A[i-k+1]$，否则暴力扫描一次。计算$p$与$s$的后缀的最长公共前缀也是类似的方法。可以证明以上过程的时间复杂度是线性的。

【接口】

void ExtendedKMP(char *a, char *b, int M, int N, int *Next, int *ret);

复杂度：$O(N+M)$

输　入：a, b　　求a关于b的后缀的最长公共前缀

M, N　　a, b的长度

$Next$　　a关于自己每个后缀的最长公共前缀

ret　　a关于b的每个后缀的最长公共前缀

输　出：结果保存在$Next$和ret中

【代码】

```
void ExtendedKMP(char *a, char *b,int M,int N,int *Next,int *ret){
    int i, j, k;
    for (j = 0; 1 + j < M && a[j] == a[1 + j]; j++);
    Next[1] = j;
    k = 1;
    for (i = 2; i < M; i++) {
        int Len = k + Next[k], L = Next[i - k];
        if (L < Len - i) {
            Next[i] = L;
        } else {
            for (j = max(0, Len - i); i + j < M && a[j] == a[i + j]; j++);
            Next[i] = j;
            k = i;
        }
    }
    for (j = 0; j < N && j < M && a[j] == b[j]; j++);
    ret[0] = j;
    k = 0;
    for (i = 1; i < N; i++) {
        int Len = k + ret[k], L = Next[i - k];
        if (L < Len - i) {
```

```
                ret[i] = L;
            } else {
                for (j = max(0, Len - i); j < M && i + j < N && a[j] ==
                b[i + j]; j++);
                ret[i] = j;
                k = i;
            }
        }
    }
```

【使用范例】

参见程序 POJ1699.CPP。

5.1.3 串的最小表示

【任务】

给定一个环形的字符串s，求字符串t，使得t是所有与s长度相同的子串里字典序最小的字符串。

【说明】

首先我们将字符串复制一遍接在原串后，将环转化为链。

我们用两个指针i和j维护最优起始位置和待比较起始位置。

令$k = \{最小的x \mid s[i + x] \neq s[j + x]\}$，如果$k \geqslant N$，那么$i$已经是最优起始位置了。否则，当$s[j + k] > s[i + k]$的时候，我们直接将$j$向后滑动$k + 1$。若$s[j + k] < s[i + k]$，令$j = \max(j, i + k) + 1$并更新最优位置$i$。

重复上面的步骤，直到$j \geqslant N$为止。

【接口】

string smallestRepresation(string s);

复杂度：$O(length)$

输　入：s　表示环串

输　出：最小表示串

【代码】

```
string smallestRepresation(string s) {
    int i, j, k, l;
    int N = s.length();
```

```
    s += s;
    for(i = 0, j = 1; j < N; ) {
        for(k = 0; k < N && s[i + k] == s[j + k]; k ++);
        if(k >= N)
            break;
        if(s[i + k] < s[j + k])
            j += k + 1;
        else {
            l = i + k;
            i = j;
            j = max(l, j) + 1;
        }
    }
    return s.substr(i, N);
}
```

【使用范例】

参见程序 SPOJ_BEADS.CPP。

5.1.4　有限状态自动机

【任务】

给定n个模式串$P_1, P_2, \cdots, P_n$，由这些模式串构造一棵Trie树，树的每个节点就是一个状态。初始时状态为根节点。对于给定的状态S以及字符ch，完成状态转移函数$f(S, ch)$，它等于最深的节点v满足$str(v)$是$str(S) + ch$的后缀。其中$str(S)$代表状态S表示的字符串。

【说明】

类似KMP算法，我们对Trie树中的每个节点v求出它的前缀指针$p[v]$，它等于最深的节点u满足$str(u)$为$str(v)$的后缀。具体的方法和KMP非常相似，请参考下面给出的代码。

【接口】

void insert(char   *s,int l,int t,int x);

复杂度：O(l)

输　入：x　　当前所在节点（传入时设为$root$）

t　　当前深度（传入时设为0）

l　　插入串的长度

s　　插入的串

void build( );　　用于建立自动机

复杂度：O(n)　　其中n为Trie树中的节点个数

int child(int x,char ch);

复杂度：O(d)　　其中d为Trie树深度

若是用一个文本串在图中逐字漫游的话，复杂度通常是均摊O(1)

输　入：x　　当前状态

　　　　ch　　用于进行转移的字符（边）

输　出：得到的新状态

【代码】

```
struct tree
{
    char ch;
    int son,next,father,danger,suffix;
};
tree a[2501];
void insert(char *s,int l,int t,int x)
{
    int i;
    if (a[x].danger)
        return;
    if (a[x].son==0)
   {
        m++;
        a[x].son=m;
        a[m].father=x;
        a[m].ch=s[t];
        if (t+1==l)
            a[m].danger=1;
        else
            insert(s,l,t+1,m);
    }
    else
    {
        i=a[x].son;
        while (1)
        {
            if (a[i].next==0 || a[i].ch==s[t])
```

```
                break;
            i=a[i].next;
        }
        if (a[i].ch==s[t] && t+1==l)
            a[i].danger=1;
        else if (a[i].ch==s[t])
            insert(s,l,t+1,i);
        else
        {
            m++;
            a[i].next=m;
            a[m].father=x;
            a[m].ch=s[t];
            if (t+1==l)
                a[m].danger=1;
            else
               insert(s,l,t+1,m);
        }
    }
}
void build()
{
    int child(int,char);
    int i,l,r;
    l=r=1;
    q[1]=1;
    a[1].suffix=1;
    if (a[1].son==0)
        return;
    while (l<=r)
    {
        if (!a[q[l]].danger)
        {
            i=a[q[l]].son;
            while (1)
            {
                r++;
                q[r]=i;
                i=a[i].next;
```

```
                if (i==0)
                    break;
            }
        }
        l++;
    }
    for (i=2;i<=r;i++)
    {
        if (a[q[i]].father==1)
        {
            a[q[i]].suffix=1;
            continue;
        }
        a[q[i]].suffix=child(a[a[q[i]].father].suffix,a[q[i]].ch);
        if (a[a[q[i]].suffix].danger)
            a[q[i]].danger=1;
    }
}
int child(int x,char ch)
{
    int i;
    i=a[x].son;
     while (i!=0)
    {
        if (a[i].ch==ch)
            break;
        i=a[i].next;
    }
    if (i!=0)
        return (i);
    else if (x==1)
        return (1);
    else
        return (child(a[x].suffix,ch));
}
```

【使用范例】

参见程序 URAL1158.CPP。

5.1.5 后缀数组

【任务】

给定一个字符串S，长度为n，设$S(i)$表示S的长度为i的后缀。给所有$S(i)$排序。严格地说，求出一个0到$n-1$的排列P，使得$S(P[0]) < S(P[1]) < \cdots < S(P[n-1])$。$P$就是$S$的后缀数组。

【说明】

倍增算法的基本思想是，设$S[i,j]$表示S从第i位开始，连续j个字符构成的字符串，那么我们首先对所有$S[i,1]$排序，然后利用上一步的结果，对所有的$S[i,2]$排序。接下来是$S[i,4], S[i,8], \cdots$,一直到$S[i,2k]$（$2k \geqslant n$）此时我们就完成了对所有后缀的排序。

具体实现如下：假设我们现在对所有$S[i,2i]$排序，$rank[i]$代表$S[i,2i-1]$在排序之后排在第几位（即它是第几大的）。对于所有i构造一个二元组（$rank[i], rank[i+2k-1]$），对所有的二元组排序就相当于对$S[i,2i]$排序（可以理解为将$S[i,2i]$视为2个字符）。由于所有的$rank$值都不大于n，我们采用基数排序，时间复杂度为$\mathrm{O}(n)$。由于一共需要进行$\log n$次排序，总的时间复杂度为$\mathrm{O}(n\log n)$。

后缀数组的一个重要应用是可以利用后缀数组快速地求出两个后缀$S(i), S(j)$的最长公共前缀（LCP, Longest Common Prefix）。做法如下：

定义$h[i]$代表$S(sa[i])$与$S(sa[i-1])$的最长公共前缀，那么$S(i), S(j)$（设$sa[i] < sa[j]$）的最长公共前缀就是$h[sa[i]+1], h[sa[i]+2], \cdots, h[sa[j]]$中的最小值。

$h[]$如果根据定义暴力计算时间复杂度过高，但基于以下事实，$h[]$的计算可以做到$\mathrm{O}(n)$：$h[rank[i]] \geqslant h[rank[i-1]]-1$。

考虑后缀$S(i-1)$与排在它前一位的后缀$S(sa[rank[i-1]-1])$的最长公共前缀，把这个最长公共前缀的首字符去掉，就是后缀$S(i)$与$S(sa[rank[i]-1])$的一个公共前缀，所以上面那个不等式成立。

【接口】

void suffix_array(int *str, int *sa, int n, int m);

复杂度：$\mathrm{O}(n\log n)$

输 入：str 字符串

n 字符串的长度

m 字符串中最大的字符

sa str的后缀数组

输 出：函数结束后sa数组即为str的后缀数组

void calc_height(int *str, int *sa, int *h, int n);

输　入：*str*　　字符串

sa　　该字符串的后缀数组

n　　字符串长度

h　　上述的*h*数组

输　出：计算*h*[]，并保存在*h*数组中

【代码】

```
void radix(int *str, int *a, int *b, int n, int m) {
    static int count[200000];
    memset(count, 0, sizeof(count));
    for (int i = 0; i < n; ++i) ++count[str[a[i]]];
    for (int i = 1; i <= m; ++i) count[i] += count[i-1];
    for (int i = n-1; i >= 0; --i) b[--count[str[a[i]]]] = a[i];
}

void suffix_array(int *str, int *sa, int n, int m) {
    static int rank[200000], a[200000], b[200000];
    for (int i = 0; i < n; ++i) rank[i] = i;
    radix(str, rank, sa, n, m);

    rank[sa[0]] = 0;
    for (int i = 1; i < n; ++i) rank[sa[i]] = rank[sa[i-1]] + (str[sa[i]]
        != str[sa[i-1]]);
    for (int i = 0; 1<<i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            a[j] = rank[j] + 1;
            b[j] = j + (1<<i) >= n ? 0 : rank[j + (1 << i)] + 1;
            sa[j] = j;
        }
        radix(b, sa, rank, n, n);
        radix(a, rank, sa, n, n);
        rank[sa[0]] = 0;
        for (int j = 1; j < n; ++j) {
            rank[sa[j]] = rank[sa[j-1]] + (a[sa[j-1]] != a[sa[j]] ||
            b[sa[j-1]] != b[sa[j]]);
        }
    }
```

```
}

void calc_height(int *str, int *sa, int *h, int n) {
    static int rank[200000];
    int k = 0;
    h[0] = 0;
    for (int i = 0; i < n; ++i) rank[sa[i]] = i;
    for (int i = 0; i < n; ++i) {
        k = k == 0 ? 0 : k - 1;
        if (rank[i] != 0)
            while (str[i + k] == str[sa[rank[i]-1] + k]) ++k;
        h[rank[i]] = k;
    }
}
```

【使用范例】

参见程序 URAL1517.CPP。

5.1.6　最长重复子串

【任务】

给定一个字符串S，求出：

（1）最长的S的子串，满足它在S中出现了至少两次；

（2）最长的S的子串，满足它在S中出现了至少两次，且不互相重叠。

【说明】

我们使用后缀数组来解决这个问题。第一个问题比较简单，求出所有$Height$值，取最大值即可。

考虑第二个问题，我们首先二分答案Ans，然后利用它对所有后缀进行分组：对于两个相邻的后缀S_i和S_{i+1}，如果它们的最长公共前缀大于等于Ans，那么将它们分为一组。很明显，同一组之内的后缀两两的最长公共前缀不小于Ans。检查每一组后缀，如果其中存在两个后缀的位置之差大于Ans，那么说明答案Ans是可行的。

【接口】

string duplicate_substr(string str, int kind);

复杂度：后缀数组复杂度

输　入：*str*　　字符串

kind 所求问题的种类

输 出：$kind = 1$时：返回最长的str的子串，满足它在str中出现了至少两次；

$kind = 2$时：返回最长的str的子串，满足它在str中出现了至少两次，且不互相重叠。

调用外部程序：

后缀数组：参见 5.1.5 节

【代码】

```
string duplicate_substr(string str, int kind) {
    string rev;
    static int s[3000], sa[3000], rank[3000], h[3000];
    int n = str.length();

    copy(str.begin(), str.end(), s);
    suffix_array (s, sa, n, 256);

    for (int i = 0; i < n; ++i) {
        rank[sa[i]] = i;
    }

    int k = 0;
    int ans1 = 0, pos1 = 0;
    for (int i = 0; i < n; ++i) {
        k = k == 0 ? 0 : k - 1;
        while (rank[i] > 0 && s[i + k] == s[sa[rank[i]-1] + k]) {
            ++k;
        }
        h[rank[i]] = k;
        if (h[rank[i]] > ans1) {
            ans1 = h[rank[i]];
            pos1 = i;
        }
    }
     if (kind == 1)
    return str.substr(pos1, ans1);

    int low = 1, high = n;
    int ans2 = 0, pos21 = 0, pos22 = 0;
    while (low <= high) {
```

```
        int mid = (low + high) / 2;
        bool ok = false;
        for (int i = 0; i < n;) {
            int j = i + 1, minPos = sa[i], maxPos = sa[i];
            while (j < n && h[j] >= mid) {
                minPos = min(minPos, sa[j]);
                maxPos = max(maxPos, sa[j]);
                ++j;
            }
            if (maxPos - minPos >= mid) {
                ok = true;
                if (mid > ans2) {
                    ans2 = mid;
                    pos21 = minPos;
                    pos22 = maxPos;
                }
                break;
            }
            i = j;
        }
        if (ok) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    if (kind == 2)
        return str.substr(pos21, ans2);
}
```

【使用范例】

参见程序 POJ1743.CPP。

5.1.7 最长公共子串

【任务】

给定两个字符串S_1, S_2，求出它们的最长公共子串。

【说明】

将S_1,S_2连接成一个字符串，中间用一个最小的不出现在S_1,S_2中的字符隔开，求出新字符串的后缀数组。如果两个相邻后缀不同时属于S_1或者S_2，那么它们的最长公共前缀就是一个公共子串。在所有公共子串中取最大值即可。

【接口】

int work(string a, string b);

复杂度：后缀数组复杂度

输　入：a,b 两个字符串

输　出：a,b的最长公共子串

调用外部程序：

　　后缀数组：参见 5.1.5 节

【代码】

```
const int MAXN = 200005;

int work(string a, string b) {
    static int s[MAXN], sa[MAXN], h[MAXN], rank[MAXN];
    string str;
    str = a + "#" + b;
    copy(str.begin(), str.end(), s);

    suffix_array(s, sa, str.length(), str.length() + 256);
    for (int i = 0; i < str.length(); ++i) {
        rank[sa[i]] = i;
    }
    int curH = 0;
    for (int i = 0; i < str.length(); ++i) {
        curH = curH == 0 ? 0 : curH - 1;
        if (rank[i] != 0) {
            while (str[i + curH] == str[sa[rank[i]-1] + curH]) {
                ++curH;
            }
        } else {
            curH = 0;
        }
        h[rank[i]] = curH;
```

```
    }

    int ans = 0, pos;
    for (int i = 1; i < str.length(); ++i) {
    if (h[i] > ans && (sa[i-1] < a.length()) != (sa[i] < a.length())) {
            ans = h[i];
            pos = sa[i];
        }
    }
    if (ans == 0) {
        cout << "Not Found" << endl;
    } else {
        cout << str.substr(pos, ans) << endl;
    }
    return 0;
}
```

【使用范例】

参见程序 URAL1517.CPP。

5.1.8　最长回文子串 manacher 算法

【任务】

给定一个字符串S，求出S的最长回文子串。

【说明】

为了方便处理回文串奇偶两种情况，我们把位置在$[i,j]$的回文串的长度信息存储在$len[i+j]$的位置上。类似扩展 KMP，假设现在要计算$len[i]$，设j满足$j < i$且$r = \left\lfloor \frac{j+1}{2} \right\rfloor + len[j] - 1$最大，根据对称性可以知道，$len[i]$至少是$\min\left\{ len\left[\left\lfloor \frac{j}{2} \right\rfloor - i\right], r - \left\lfloor \frac{i+1}{2} \right\rfloor \right\}$，之后暴力匹配即可。

【接口】

void find_palindrome(char str[], int len[], int n);

复杂度：$O(n)$

输　入：str　文本串

　　　　n　文本串长度
输　出：len　所有中心的回文串长度

【代码】

```
void manacher(char str[], int len[], int n) {
    len[0] = 1;
    for (int i = 1, j = 0; i < (n << 1) - 1; ++i) {
        int p = i >> 1, q = i - p, r = ((j + 1) >> 1) + len[j] - 1;
        len[i] = r < q ? 0 : min(r - q + 1, len[(j << 1) - i]);
        while (p > len[i] - 1 && q + len[i] < n && str[p - len[i]]
        == str[q + len[i]])
            ++len[i];
        if(q + len[i] - 1 > r)
            j = i;
    }
}
```

【使用范例】

参见程序 URAL1297.CPP。

5.1.9　字符串散列

【任务】

要求为一个字符串设计一种散列。

【说明】

一种比较常见的方法是对于第i个字符，让它对散列的贡献值表示成$s[i] \times P^i$，其中P为一个素数（为了方便，这里的第i个指的是从右往左数）。

【接口】

void init_hash(int L, char　*s, unsigned int　*h);
复杂度：$O(L)$
输　入：L　字符串长度
　　　　s　字符串
　　　　h　串S的散列值预处理到h中

unsigned int string_hash(unsigned int　*h, int l, int r);
复杂度：$O(1)$

输　入：h　散列值预处理结果
　　　　l,r　需要散列的子串的首尾下标
输　出：子串$s[l,r)$的散列值（编号从0开始）

【代码】

```
inline void init_hash(int l, char *s, unsigned int *h) {
    h[0] = 0;
    for (int i = 1; i <= l; ++i)
        h[i] = h[i - 1] * MAGIC + s[i - 1];
    base[0] = 1;
    for (int i = 1; i <= l; ++i)
        base[i] = base[i - 1] * MAGIC;
}

inline unsigned int string_hash(unsigned int *h, int l, int r) {
    return h[r] - h[l] * base[r - l];
}
```

【使用范例】

参见程序 POJ2503.CPP。

5.2　转　换

5.2.1　星期计算

【任务】

给定一个日期，问这个日期是星期几。

【说明】

第一个方法可以计算这个日期与今天的距离X，假设今天是星期y,那么给定日期就是星期$((y-X)\%7+7)\%7+1$（如果给定日期是今天之前的日期），或者星期$(y+X)\%7+1$。（给定日期是未来的日期）

第二个方法是直接使用蔡勒公式：

$$Week=(Day+2\times Month\times 3\times(Month+1)/5+Year+Year/4-Year/100+Year/400)\%7$$

当日期在 1752 年 9 月 3 日之前时：

$$Week=(Day+2\times Month+3\times(Month+1)/5+Year+Year/4+5)\%7$$

【接口】

int whatday(int d, int m, int y);

复杂度：O(1)

输　入：d, m, y　日期的日、月、年

输　出：返回ans，表示是星期$(ans + 1)$

【代码】

```
int whatday(int d, int m, int y)
{
    int ans;
    if(m==1 || m==2)
        m += 12, y--;
    if((y<1752)||(y==1752&&m<9)||(y==1752&&m==9&&d<3))
        ans = (d+2*m+3*(m+1)/5 + y + y/4+5)%7;
    else
        ans = (d+2*m+3*(m+1)/5 + y + y/4 - y/100 + y/400)%7;
    return ans;
}
```

【使用范例】

参见程序 SWUST078.CPP。

【注解】

罗马教皇格里高利 13 世在 1582 年组织了一批天文学家，根据哥白尼日心说计算出来的数据，对儒略历作了修改。将 1582 年 10 月 5 日到 14 日之间的 10 天宣布撤销，继 10 月 4 日之后为 10 月 15 日。后来人们将这一新的历法称为“格里高利历”，也就是今天世界上所通用的历法，简称格里历或公历。不同国家用取消旧历法启用新历法的年代不同，导致蔡勒公式的不同版本。

实际使用的时候请注意遵循题目描述的规则。

5.2.2　日期相隔天数计算

【任务】

给定 2 个日期A,B，求A, B间相隔了多少天。

【说明】

计算公元元年到A和B分别有多少天，然后两个值相减即可。

【接口】

int count_day(int da, int ma, int ya, int db, int mb, int yb);

复杂度：O(1)

输　入：da, ma, ya　　A的日、月、年

　　　　db, mb, yb　　B的日、月、年

输　出：A, B间的相隔天数

【代码】

```
const int days = 365;
const int s[] = {0,31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

bool Isleap(int y)
{
    if(y%400==0 || y%100 && y%4==0)return 1;
    return 0;
}

int leap(int y)
{
    if(!y)return 0;
    return y/4-y/100+y/400;
}

int calc(int day, int mon, int year)
{
    int res = (year-1) * days + leap(year-1);
    for(int i = 1; i < mon; ++i)
        res += s[i];
    if(Isleap(year) && mon > 2) res++;
    res += day;
    return res;
}

int count_day(int da, intma, intya, intdb, int mb,int yb)
{
    int resa = calc(da, ma, ya);
    int resb = calc(db, mb, yb);
    return abs(resa-resb);
}
```

【注解】

不考虑公元前以及历史上的日历调整。

【使用范例】

参见 DaysBetweenDates.CPP。

5.2.3 斐波那契进制转换

【任务】

给定一个正整数n，把它写成类似二进制的形式，使得$n = \sum fib[i] \times a[i]$，其中$a[i] = 0$或1且1的个数尽量少。其中$fib[i]$表示斐波那契数列第$i$项。

【说明】

从高到低枚举i，如果当前$fib[i] \leqslant n$，那么$a[i] = 1$，并且$n = n - fib[i]$，否则$a[i] = 0$。可以证明上述方法能使得每个n都能被表示出来，并且是唯一的。

【接口】

vector<int> solve(int n);

复杂度：O(log$n$)

输　入：n　要转换的数

输　出：n的斐波那契进制表示

【代码】

```
const int maxn = 300;

int fib[maxn];
int a[maxn], lim;

vector<int> solve(int n) {
    fib[0] = fib[1] = 1;
    for(int i = 2; i < maxn; ++i)
    {
        fib[i] = fib[i-1] + fib[i-2];
        if(fib[i] > n)
        {
            lim = i;
            break;
```

```
            }
        }
        vector<int> ret;
        for(int i = lim-1; i > 0; --i)
            if(fib[i] <= n)
            {
                ret.push_back(1);
                n -= fib[i];
            } else ret.push_back(0);
        return ret;
    }
```

【使用范例】

参见程序 UVA948.CPP。

5.2.4　罗马进制转换

【任务】

给定一个正的十进制数，将其转换成罗马数字。

【说明】

首先要了解罗马数字的规则，它由7个基本数字组成。

I：1　V：5　X：10　L：50　C：100　D：500　M：1000

如果要表示9或4的时候，在符号10或5前加一个1的符号表示减去1,90,900的也类似。

做法如下：依次执行下面的步骤：

（1）如果数$a \geqslant 1000$，那么输出$\left\lfloor \frac{a}{1000} \right\rfloor$个 M，$a$减去$\left\lfloor \frac{a}{1000} \right\rfloor \times 1000$。

（2）如果$a \geqslant 900$，输出 CM，a减去900。

（3）如果$a \geqslant 500$，输出 D，a减去500。

（4）如果$a \geqslant 400$，输出 CD，a减去400。

（5）如果$a \geqslant 100$，输出 C，a减去100。

对于$a < 100$的情况类似处理即可。

【接口】

string rome(int a);

复杂度：$O(\log_{10} a)$

输　入：a　要转换的数

输　出：a的罗马进制表示。

【代码】

```
string rome(int a)
{
    string s;
    int i,j;
    if(a>=1000) {
        i=a/1000;
        for(j=0;j<i;j++)
            s = s + "M";
        a-=1000*i;
    }
    if(a>=900){
        s = s + "CM"; a-=900;
    }
    if(a>=500){
        s = s + "D"; a-=500;
    }
    if(a>=400){
        s = s + "CD"; a-=400;
    }
    if(a>=100){
        i=a/100;
        for(j=0;j<i;j++)
        s = s + "C";
        a-=100*i;
    }
    if(a>=90){
        s = s + "XC"; a-=90;
    }
    if(a>=50){
        s = s + "L"; a-=50;
    }
    if(a>=40) {
        s = s + "XL"; a-=40;
    }
    if(a>=10){
        i=a/10;
```

```
            for(j=0;j<i;j++)
            s = s + "X";
            a-=10*i;
        }
        if(a>=9){
            s = s + "IX"; a-=9;
        }
        if(a>=5){
            s=s+"V"; a-=5;
        }
        if(a>=4){
            s=s+"IV"; a-=4;
        }
        for(j=0;j<a;j++)
            s=s+"I";
        return s;
    }
```

【使用范例】

参见程序 USACO_PREFACE.CPP。

5.3　构　　造

5.3.1　幻方构造

【任务】

构造一个N阶幻方。

【说明】

幻方要求将1到N^2的数填入$N \times N$的矩阵中，使得每行、每列和两条对角线上的和相等。

【接口】

void generate(int n,int d[][MAXN]);

输入：n　构造n阶幻方

输出：d　存放构造结果

【代码】

```
void dllb(int l,int si,int sj,int sn,int d[][MAXN]){
    int n,i=0,j=l/2;
```

```
    for (n=1;n<=l*l;n++){
        d[i+si][j+sj]=n+sn;
        if (n%l){
            i=(i)?(i-1):(l-1);
            j=(j==l-1)?0:(j+1);
        }
        else
            i=(i==l-1)?0:(i+1);
    }
}

void magic_odd(int l,int d[][MAXN]){
    dllb(l,0,0,0,d);
}

void magic_4k(int l,int d[][MAXN]){
    int i,j;
    for (i=0;i<l;i++)
        for (j=0;j<l;j++)
            d[i][j]=(((i%4==0||i%4==3)&&(j%4==0||j%4==3))||
            ((i%4==1||i%4==2)&&(j%4==1||j%4==2)))?(l*l-(i*l+j)):
            (i*l+j+1);
}

void magic_other(int l,int d[][MAXN]){
    int i,j,t;
    dllb(l/2,0,0,0,d);
    dllb(l/2,l/2,l/2,l*l/4,d);
    dllb(l/2,0,l/2,l*l/2,d);
    dllb(l/2,l/2,0,l*l/4*3,d);
    for (i=0;i<l/2;i++)
        for (j=0;j<l/4;j++)
            if (i!=l/4||j)
                t=d[i][j],d[i][j]=d[i+l/2][j],d[i+l/2][j]=t;
    t=d[l/4][l/4],d[l/4][l/4]=d[l/4+l/2][l/4],d[l/4+l/2][l/4]=t;
    for (i=0;i<l/2;i++)
        for (j=l-l/4+1;j<l;j++)
            t=d[i][j],d[i][j]=d[i+l/2][j],d[i+l/2][j]=t;
}
```

```
void generate(int n,int d[][MAXN]){
    if (n%2)
        magic_odd(l,d);
    else if (n%4==0)
        magic_4k(n,d);
    else
        magic_other(n,d);
}
```

【注释】

$N = 2$时幻方是无解的。

【使用范例】

参见程序 MagicSquare.CPP。

5.3.2　N 皇后问题

【任务】

在$n \times n$的棋盘上放 n 个皇后，使得它们互相不能攻击。

【说明】

令$k = n \text{ div } 2$，讨论n的情况：

若$n \bmod 6 \neq 2$且$n \bmod 6 \neq 3$：

n是偶数则有：$2, 4, 6, 8, \cdots, n, 1, 3, 5, 7, \cdots, n-1$

n是奇数则有：$2, 4, 6, 8, \cdots, n-1, 1, 3, 5, 7, \cdots, n$

若$n \bmod 6 = 2$或$n \bmod 6 = 3$：

k为偶数，n为偶数:

$k, k+2, k+4, \cdots, n, 2, 4, \cdots, k-2, k+3, k+5, \cdots, n-1, 1, 3, 5, \cdots, k+1$

k为偶数，n为奇数:

$k, k+2, k+4, \cdots, n-1, 2, 4, \cdots, k-2, k+3, k+5, \cdots, n-2, 1, 3, 5, \cdots, k+1, n$

k为奇数，n为偶数:

$k, k+2, k+4, \cdots, n-1, 1, 3, 5, \cdots, k-2, k+3, \cdots, n, 2, 4, \cdots, k+1$

k为奇数，n为奇数:

$k, k+2, k+4, \cdots, n-2, 1, 3, 5, \cdots, k-2, k+3, \cdots, n-1, 2, 4, \cdots, k+1, n$

【接口】

void solve_nqueen(int n);

输入：n　棋盘规模

输出：n个数，第i位上的数j代表第i行的皇后放在左数第j个格子里

【代码】

```
void solve_nqueen(int n) {
    int k;
    first = true;
    if (n % 6 != 2 && n % 6 != 3) {
        for (int i = 2; i <= n; i += 2)
            Print(i);
        for (int i = 1; i <= n; i += 2)
            Print(i);
    } else {
        k = n / 2;
        if (k % 2 == 0) {
            for (int i = k; i <= n; i += 2)
                Print(i);
            for (int i = 2; i <= k - 2; i += 2)
                Print(i);
            for (int i = k + 3; i <= n - 1; i += 2)
                Print(i);
            for (int i = 1; i <= k + 1; i += 2)
                Print(i);
            if (n % 2 == 1)
                Print(n);
        } else {
            for (int i = k; i <= n - 1; i += 2)
                Print(i);
            for (int i = 1; i <= k - 2; i += 2)
                Print(i);
            for (int i = k+3; i <= n; i += 2)
                Print(i);
            for (int i = 2; i <= k+1; i += 2)
                Print(i);
            if (n % 2 == 1)
                Print(n);
```

```
        }
    }
    printf("\n");
}
```

【注释】

程序中 Print 过程是输出一个数。你可以把它替换成其他操作比如加入一个 vector。

【使用范例】

参见程序 POJ3239.CPP。

5.3.3　旋转魔方

【任务】

模拟一个三阶魔方的旋转操作。读入一个魔方，输出按一定步骤旋转后的结果。

【说明】

魔方的每个面都存放在长度为9的一维数组中。下图为魔方每个面上每个位置的编号，其中正中间的面为魔方正面（即面向自己的面），如标注所示。

			0	1	2						
			3	4	5	up					
			6	7	8						
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
6	7	8	6	7	8	6	7	8	6	7	8
	left		0	1	2		right			back	
			3	4	5	down					
			6	7	8						

魔方的六个面分别用六个字母表示，F=front，B=back，U=up，D=down，L=left，R=right。

六个函数L,R,U,D,F,B分别表示对每个面进行顺时针旋转的操作。参数cnt为执行旋转的次数，默认为1，若要逆时针旋转令$cnt = 3$即可。

每个格子上默认用一个char表示颜色信息，也可以修改成其他类型。

该代码不会检验读入数据的合法性。

【接口】

结构体：*MagicCube*

成员变量：

Elem $f[9], b[9], u[9], d[9], l[9], r[9]$ 存放6个面的信息

成员函数：

L( ),R( ),U( ),D( ),F( ),B( )	6种模仿基本旋转操作
void operate(string str);	执行*str*操作指令序列 大写的字母表示对一个面顺时针旋转 小写字母表示逆时针旋转
void print( );	打印出每个面的信息

【代码】

```
struct MagicCube{
    typedef char Elem;              //用字符表示每格的颜色
    Elem f[9],b[9],u[9],d[9],l[9],r[9],tmp[9],ch;
    // 读入 6 个面
    void get_up(){ for (int i=0;i<9;i++) cin>>u[i]; }
    void get_front(){ for (int i=0;i<9;i++) cin>>f[i]; }
    void get_back(){ for (int i=0;i<9;i++) cin>>b[i]; }
    void get_left(){ for (int i=0;i<9;i++) cin>>l[i]; }
    void get_right(){ for (int i=0;i<9;i++) cin>>r[i]; }
    void get_down(){ for (int i=0;i<9;i++) cin>>d[i]; }
    // 旋转面
    int right_rotate(Elem a[9]){
        for (int i=0;i<9;i++) tmp[i]=a[i];
        for (int i=0;i<3;i++)
            for (int j=0;j<3;j++)
                a[i*3+j]=tmp[3*(2-j)+i];
    }
    // 基本操作
    void L(int cnt=1){
        for (;cnt>0;cnt--){
            right_rotate(l);
            ch=u[0],u[0]=b[8],b[8]=d[0],d[0]=f[0],f[0]=ch;
            ch=u[3],u[3]=b[5],b[5]=d[3],d[3]=f[3],f[3]=ch;
            ch=u[6],u[6]=b[2],b[2]=d[6],d[6]=f[6],f[6]=ch;
        }
    }
```

```
void R(int cnt=1){
    for (;cnt>0;cnt--){
        right_rotate(r);
        ch=u[2],u[2]=f[2],f[2]=d[2],d[2]=b[6],b[6]=ch;
        ch=u[5],u[5]=f[5],f[5]=d[5],d[5]=b[3],b[3]=ch;
        ch=u[8],u[8]=f[8],f[8]=d[8],d[8]=b[0],b[0]=ch;
    }
}
void U(int cnt=1){
    for (;cnt>0;cnt--){
        right_rotate(u);
        ch=f[0],f[0]=r[0],r[0]=b[0],b[0]=l[0],l[0]=ch;
        ch=f[1],f[1]=r[1],r[1]=b[1],b[1]=l[1],l[1]=ch;
        ch=f[2],f[2]=r[2],r[2]=b[2],b[2]=l[2],l[2]=ch;
    }
}
void D(int cnt=1){
    for (;cnt>0;cnt--){
        right_rotate(d);
        ch=f[6],f[6]=l[6],l[6]=b[6],b[6]=r[6],r[6]=ch;
        ch=f[7],f[7]=l[7],l[7]=b[7],b[7]=r[7],r[7]=ch;
        ch=f[8],f[8]=l[8],l[8]=b[8],b[8]=r[8],r[8]=ch;
    }
}
void F(int cnt=1){
    for (;cnt>0;cnt--){
        right_rotate(f);
        ch=u[6],u[6]=l[8],l[8]=d[2],d[2]=r[0],r[0]=ch;
        ch=u[7],u[7]=l[5],l[5]=d[1],d[1]=r[3],r[3]=ch;
        ch=u[8],u[8]=l[2],l[2]=d[0],d[0]=r[6],r[6]=ch;
    }
}
void B(int cnt=1){
    for (;cnt>0;cnt--){
        right_rotate(b);
        ch=u[0],u[0]=r[2],r[2]=d[8],d[8]=l[6],l[6]=ch;
        ch=u[1],u[1]=r[5],r[5]=d[7],d[7]=l[3],l[3]=ch;
        ch=u[2],u[2]=r[8],r[8]=d[6],d[6]=l[0],l[0]=ch;
    }
}
```

```
    void operate(string str){
        for (int i=0;i<str.length();i++)
            if (isupper(str[i])){
                if (str[i]=='L') L(1);
                if (str[i]=='R') R(1);
                if (str[i]=='U') U(1);
                if (str[i]=='D') D(1);
                if (str[i]=='F') F(1);
                if (str[i]=='B') B(1);
            }else
            if (islower(str[i])){
                if (str[i]=='l') L(3);
                if (str[i]=='r') R(3);
                if (str[i]=='u') U(3);
                if (str[i]=='d') D(3);
                if (str[i]=='f') F(3);
                if (str[i]=='b') B(3);
            }
    }
    // 输出魔方
    void print(){
        for (int i=0;i<9;i++) cout<<f[i]; cout<<endl;
        for (int i=0;i<9;i++) cout<<d[i]; cout<<endl;
        for (int i=0;i<9;i++) cout<<l[i]; cout<<endl;
        for (int i=0;i<9;i++) cout<<r[i]; cout<<endl;
        for (int i=0;i<9;i++) cout<<u[i]; cout<<endl;
        for (int i=0;i<9;i++) cout<<d[i]; cout<<endl;
        cout<<endl;
    }
};
```

【使用范例】

参见程序 POJ1955.CPP。

5.3.4 骑士周游问题

【任务】

给定一个国际象棋棋盘，问国际象棋的马是否能走出一条路，使得每个格子都恰被访问一次。

【说明】

这道题本质上是个哈密顿链问题，没有很好的多项式算法，只能搜索。

但是有一个优化：求出对于每个格子x，最多能有多少个格子走到x上去，设为$vis[x]$。我们可以按照$vis[x]$从小到大排序，每次走到下一步时选$vis[x]$最小的格子走。

同时表示已访问格子的状态可以使用位压缩的方法进行加速。

有了这两个优化后，解已经可以非常迅速地找出来。

【接口】

vector<pair<int,int> > solve(int NX0, int NY0);

输入：$NX0, NY0$　棋盘大小

输出：返回一个vector用pair<int,int>存有$NX0 \times NY0$个坐标，表示骑士周游的路线

【代码】

```
#define two(X) ((ULL)1<<(X))

typedef unsigned long long ULL;
const int dx[] = {2, 1, -1, -2, -2, -1, 1, 2};
const int dy[] = {-1, -2, -2, -1, 1, 2, 2, 1};
ULL lim = two(63)-1 + two(63);
int cnt[8][8], NX, NY;
vector<pair<int,int> > answ;

bool dfs(int x, int y, ULL state)
{
    if(state==lim)
    {
        answ.push_back(make_pair(x,y));
        return 1;
    }

    int ct = 0;
    int px[9], py[9], id[9];
    for(int i = 0; i < 8; ++i)
    {
        int nx = x+dx[i];
        int ny = y+dy[i];
        if(nx>=0&&nx<NX&&ny>=0&&ny<NY&&!(state&two(nx*NY+ny)))
        {
```

```
                px[ct] = nx; py[ct] = ny;
                ct++;
            }
        }
        if(!ct)return 0;
        for(int i = 0; i < ct; ++i)id[i] = i;
        for(int i = 0; i < ct; ++i)
        for(int j = i+1; j < ct; ++j)
        if(cnt[px[id[i]]][py[id[i]]] > cnt[px[id[j]]][py[id[j]]])
            swap(id[i], id[j]);

        for(int i = 0; i < ct; ++i)
        if(dfs( px[id[i]], py[id[i]], state|two(px[id[i]]*NY+py[id[i]]) ))
        {
            answ.push_back(make_pair(x,y));
            return 1;
        }
        return 0;
}

vector<pair<int,int> > solve(int NX0, int NY0)
{
        NX=NX0; NY=NY0;
        answ.clear();
        lim = two(NX * NY - 1) - 1 + two(NX * NY - 1);
        for(int i = 0; i < 8; ++i)
        for(int j = 0; j < 8; ++j)
        {
            cnt[i][j] = 0;
            for(int k = 0; k < 8; ++k)
            {
                int nx = i+dx[k];
                int ny = j+dy[k];
                if(nx>=0&&nx<NX&&ny>=0&&ny<NY)
                cnt[i][j]++;
            }
        }
        dfs(0, 0, 1);
        return answ;
}
```

【使用范例】

参见程序 POJ2488.CPP。

5.4 计　　算

5.4.1 表达式计算

【任务】

给一个中缀表达式，计算表达式的值。表达式中包含实数，运算符，括号以及由大写或小写字母表示的变量。

【说明】

int *priv*[]数组记录运算符的优先级，其中求一个数的幂优先级最高，其次是乘除，再其次是加减。有括号则括号优先级更高。

double *val*[]数组记录表达式中用到的变量的值。变量都用一个大写或小写的字母表示。

stack <double> *num*用于记录运算数的栈。

stack <char> *oper*记录运算符的栈。

用两个栈处理表达式。从左往右扫描读入的表达式，遇到数值或变量就放入*num*栈，遇到运算符则先弹出*oper*栈顶优先级大于等于自己的元素（注意这里默认运算符是左结合的话，如果碰到连续的右结合运算符比如'^'，判断的时候就不能加等号），然后将运算符加入*oper*栈中。

弹出操作，就是弹出*num*顶端两个运算数和*oper*栈顶运算符，计算后的值加入*num*栈。

括号特殊处理。栈中的左括号不会被运算符弹出，只有遇到右括号才可以消掉一个栈顶的左括号。变量或数值前的负号同样特殊处理，在负号之前加一个0。

【接口】

double calculate(string str, double val[ ]);

复杂度：O(n)

输　入：*str*　　需计算的合法表达式

　　　　val[ ]　　表达式中用到的变量的值

输　出：表达式的运算结果

【代码】

```
int priv[300];
double value[300];
double calc(double a,double b,char op){
```

```
    if (op=='+') return a+b;
    if (op=='-') return a-b;
    if (op=='*') return a*b;
    if (op=='/') return a/b;
    if (op=='^') return exp(log(a)*b);
}
double calculate(string str, double val[]=value){
    stack <double> num;
    stack <char> oper;
    priv['+']=priv['-']=3;
    priv['*']=priv['/']=2;
    priv['^']=1, priv['(']=10;
    double x,y,t=0;
    int i; char last=0;
    for (i=0;i<str.length();i++){
        if (isalpha(str[i])){
            num.push(val[str[i]]);
        }else if (isdigit(str[i])){
            num.push(atof(str.c_str()+i));
            for (;i+1<str.length()&&isdigit(str[i+1]);i++);
            if (i+1<str.length()&&str[i+1]=='.')
                for (i++;i+1<str.length()&&isdigit(str[i+1]);i++);
        }else if (str[i]=='('){
            oper.push(str[i]);
        }else if (str[i]==')'){
            while (oper.top()!='('){
                y=num.top(); num.pop();
                x=num.top(); num.pop();
                char op=oper.top();
                oper.pop();
                num.push(calc(x,y,op));
            }
            oper.pop();
        }else if (str[i]=='-'&&(last==0||last=='(')){
            num.push(0.0);
            oper.push('-');
        }else if (priv[str[i]]>0){
            while (oper.size()>0&&priv[str[i]]>=priv[oper.top()]){
                y=num.top(); num.pop();
```

```
                x=num.top(); num.pop();
                char op=oper.top();
                oper.pop();
                num.push(calc(x,y,op));
            }
            oper.push(str[i]);
        }else continue;
        last=str[i];
    }
    while (oper.size()>0){
        y=num.top(); num.pop();
        x=num.top(); num.pop();
        char op=oper.top();
        oper.pop();
        num.push(calc(x,y,op));
    }
    return num.top();
}
```

【使用范例】

参见程序 POJ1686.CPP。

5.4.2　最大权子矩形

【任务】

给一个矩阵，矩阵中每格都填有一个整数（可以是负值）。现在要求它的一个子矩阵，使得子矩阵中所有元素之和最大（或最小）。

【说明】

用 3 个数组记录数据：

（1）$a[i][j]$存放初始的矩阵i行j列的元素。

（2）$sum[i][j]$存放第j列元素$a[1][j]$到$a[i][j]$的元素之和。

（3）$arr[i]$转成一维问题后用来求解。

以求最大权的为例，先预处理出$sum[i][j]$，然后枚举子矩阵的上下界。对上下界之间的每一列求和看做一个元素，这可用已求出的sum数组快速得到，这样就将二维的问题转化为一维的最大子区间问题。

我们将前i列元素之和求出存放在$arr[i]$中，于是需要找一对$0 \leqslant i < j \leqslant m$使$arr[j]$ −

$arr[i]$的值最大。枚举j，然后用$mini$维护$arr[0]$到$arr[j-1]$之间的最小值，$arr[j]-mini$便是对于j的最大子矩阵。

求最小权，只需将初始矩阵中所有元素取相反数，做一次求最大权，答案取其相反数即可。

【接口】

int find_max(int a[N][M],int n,int m);

复杂度：$O(n^3)$

输　入：a　　矩阵

n, m　　矩阵大小

输　出：最大权子矩阵

int find_min(int a[N][M],int n,int m);　　最小权子矩阵，接口同上

【代码】

```
const int N=301,M=301;
const int inf=0x3fffffff;
int sum[N][M],arr[M];
int find_max(int a[N][M],int n,int m){
    if (n==0||m==0) return 0;
    int i,j,up,down,ret=-inf;
    memset(sum,0,sizeof(sum));
    for (i=1;i<=n;i++)
        for (j=1;j<=m;j++)
            sum[i][j]=sum[i-1][j]+a[i][j];
    arr[0]=0;
    for (up=1;up<=n;up++)
        for (down=up;down<=n;down++){
            for (i=1;i<=m;i++)
                arr[i]=arr[i-1]+(sum[down][i]-sum[up-1][i]);
            int mini=0;
            for (i=1;i<=m;i++){
                ret=max(ret,arr[i]-mini);
                mini=min(mini,arr[i]);
            }
        }
    if (ret<0) return 0;            // 若子矩阵可取空,则需作此判断
    return ret;
}
```

```
int find_min(int a[N][M],int n,int m){
    int i,j;
    for (i=1;i<=n;i++)
        for (j=1;j<=m;j++)
            a[i][j]=-a[i][j];
    int ret=-find_max(a,n,m);
    for (i=1;i<=n;i++)
        for (j=1;j<=m;j++)
            a[i][j]=-a[i][j];
    return ret;
}
```

【使用范例】

参见程序 POJ1050.CPP。

5.4.3　矩形面积并

【任务】

给你N个矩形，求这些矩形的面积并。

【说明】

$rect$存的矩形，(x_1,y_1)表示左下角，(x_2,y_2)右上角。

把一个矩形拆成两个事件，左边的竖直边为添加事件，右边的竖直边为删除事件。

evt存的事件，x表示竖直边的x坐标，y_1,y_2表示两个端点的y坐标，add为1或−1，1为添加，−1为删除。

我们先把evt根据x从小到大排序，然后依次处理每个事件。用线段树维护，添加事件就把(y_1,y_2)插入线段树，删除操作就把(y_1,y_2)从线段树中删除。每次发现$evt[i].x > evt[i-1].x$，就把当前线段覆盖的总长度$\times(evt[i].x-evt[i-1].x)$加到总面积中。

代码中把y离散化了，便于使用线段树。

【接口】

long long area( );

复杂度：$O(n\log n)$

输　入：n　　全局变量，表示矩形数

$rect$　　全局变量保存n个矩形

其中(x_1,y_1)表示左下角，(x_2,y_2)右下角

输　出：矩形并的面积

【代码】

```
const int maxn = 100000;

struct Rectangle
{
    int x1, y1, x2, y2;
};

struct Event
{
    int x, y1, y2;
    int add;
};

struct IntervalTreeNode
{
    int count, total;
};

int n;
Rectangle rect[maxn + 1];
Event evt[maxn * 2 + 1];
IntervalTreeNode tree[(maxn + 10) << 2];
int id[maxn * 2];

bool cmp(const Event &a, const Event &b)
{
    if (a.x < b.x) return true;
    return false;
}

void up(int i, int lb, int rb)
{
    tree[i].total = tree[i << 1].total + tree[(i << 1) + 1].total;
    if (tree[i].count) tree[i].total = id[rb] - id[lb];
}

```

```
void ins(int i, int lb, int rb, int a, int b, int k)
{
    if (a == lb && b == rb) {
        tree[i].count += k;
        up(i, lb, rb);
        return;
    }
    int med = (lb + rb) >> 1;
    if (b <= med)
        ins(i << 1, lb, med, a, b, k);
    else if (a >= med)
        ins((i << 1) + 1, med, rb, a, b, k);
    else {
        ins(i << 1, lb, med, a, med, k);
        ins((i << 1) + 1, med, rb, med, b, k);
    }
    up(i, lb, rb);
}

long long area()
{
   for (int i = 0; i < n; i++) {
        id[i] = rect[i].y1;
        id[i + n] = rect[i].y2;
    }
    sort(id, id + 2 * n);
    for (int i = 0; i < 2 * n; i++) {
        rect[i].y1 = lower_bound(id, id + 2 * n, rect[i].y1) - id;
        rect[i].y2 = lower_bound(id, id + 2 * n, rect[i].y2) - id;
    }

    for (int i = 0; i < n; i++) {
        evt[i].add = 1;
        evt[i + n].add = -1;
        evt[i].x = rect[i].x1;
        evt[i + n].x = rect[i].x2;
        evt[i].y1 = evt[i + n].y1 = rect[i].y1;
        evt[i].y2 = evt[i + n].y2 = rect[i].y2;
    }
    sort(evt, evt + n * 2, cmp);
```

```
        
    long long ans = 0;
    for (int i = 0; i < 2 * n; i++) {
        if (i > 0 && evt[i].x > evt[i - 1].x) {
            ans += (long long)(evt[i].x - evt[i - 1].x) * tree[1].total;
        }
        ins(1, 0, 2 * n - 1, evt[i].y1, evt[i].y2, evt[i].add);
    }
    return ans;
}
```

【使用范例】

参见程序 POJ1389.CPP。

5.4.4 矩形并的周长

【任务】

给n个矩形每个顶点的坐标，求这些矩形并起来的图形的周长。

【说明】

算法是首先将矩形顶点的坐标离散化。然后将这些矩形沿x轴平行方向切成许多条，我们知道每一条的高度，只需再知道这一条上整个图案分割成了几个部分，便能算出这一条中所有竖直的周长的长度。用一个线段树维护当前被分割成多少部分。维护时，首先将矩形拆成上下两条线段，可以看做是高度不同的两个区间。然后这些线段从低到高排序，依次操作，若是线段矩形的下边界则做添加区间的操作，否则做删除操作。每次将一个高度的区间处理完之后计算一下当前的竖直周长的总长。

对于水平的周长，换一个方向做一遍同样的算法即可。

【接口】

int solve(vector<int> x1,vector<int> y1,vector<int> x2,vector<int> y2);

复杂度：O($n\log n$)

输　入：$x1, y1, x2, y2$ 矩形左下、右上角的坐标

输　出：矩形并的总周长

【代码】

```
#define L (s<<1)
#define R ((s<<1)+1)
```

```

const int N=10000;
struct point{
    int h,l,r,f;
    point(int h,int l,int r,int f):h(h),l(l),r(r),f(f){}
};
vector<point> que1,que2;
vector<int> vx,vy;

struct Tree{
    int left[N*8],right[N*8],count[N*8],part[N*8];
    void insert(int s,int l,int r,int ll,int rr,int delta){
        if (l==ll&&r==rr){
            count[s]+=delta;
        if (count[s]){
                part[s]=left[s]=right[s]=1;
            }else
            if (ll!=rr){
                part[s]=part[L]+part[R]-(right[L]&&left[R]);
                left[s]=left[L], right[s]=right[R];
            }else part[s]=left[s]=right[s]=0;
            return;
        }
        int m=l+r>>1;
        if (rr<=m) insert(L,l,m,ll,rr,delta); else
        if (ll>m) insert(R,m+1,r,ll,rr,delta); else{
            insert(L,l,m,ll,m,delta);
            insert(R,m+1,r,m+1,rr,delta);
        }
        if (!count[s]){
            part[s]=part[L]+part[R]-(right[L]&&left[R]);
            left[s]=left[L], right[s]=right[R];
        }
    }
} T;

bool cmp(const point &a,const point &b){
    return (a.h!=b.h?a.h<b.h:a.f>b.f);
}
```

```
int solve(vector<int> x1,vector<int> y1,vector<int> x2,vector<int> y2){
    int n=x1.size(),x,ans=0,i,j;
    for (i=0;i<n;++i){
        vx.push_back(x1[i]);
        vy.push_back(y1[i]);
        vx.push_back(x2[i]);
        vy.push_back(y2[i]);
    }
    sort(vx.begin(),vx.end());
    sort(vy.begin(),vy.end());
    vx.erase(unique(vx.begin(),vx.end()),vx.end());
    vy.erase(unique(vy.begin(),vy.end()),vy.end());
    for (i=0;i<n;i++){
        x1[i]=lower_bound(vx.begin(),vx.end(),x1[i])-vx.begin();
        x2[i]=lower_bound(vx.begin(),vx.end(),x2[i])-vx.begin();
        y1[i]=lower_bound(vy.begin(),vy.end(),y1[i])-vy.begin();
        y2[i]=lower_bound(vy.begin(),vy.end(),y2[i])-vy.begin();
        que1.push_back(point(y1[i],x1[i],x2[i],1));
        que1.push_back(point(y2[i],x1[i],x2[i],-1));
        que2.push_back(point(x1[i],y1[i],y2[i],1));
        que2.push_back(point(x2[i],y1[i],y2[i],-1));
    }
    sort(que1.begin(),que1.end(),cmp);
    sort(que2.begin(),que2.end(),cmp);
    for (i=0;i<que1.size();i=j){
        for (j=i;j<que1.size()&&que1[j].h==que1[i].h;j++)
            T.insert(1,0,vx.size(),que1[j].l,que1[j].r-1,que1[j].f);
        if (que1[i].h+1<vy.size())
            ans+=T.part[1]*2*(vy[que1[i].h+1]-vy[que1[i].h]);
    }
    for (i=0;i<que2.size();i=j){
        for (j=i;j<que2.size()&&que2[j].h==que2[i].h;j++)
            T.insert(1,0,vy.size(),que2[j].l,que2[j].r-1,que2[j].f);
        if (que2[i].h+1<vx.size())
            ans+=T.part[1]*2*(vx[que2[i].h+1]-vx[que2[i].h]);
    }
    return ans;
}
```

【使用范例】

参见程序 POJ1177.CPP。

5.5 序　　列

5.5.1 第 k 小数

【任务】

给一个整数序列和一个k，求这个序列中第k小的数。

【说明】

从序列中取一个数mid，然后把序列分成小于等于mid和大于等于mid的两部分，由两个部分的元素个数和k的大小关系可以确定这个数是在哪个部分。对部分序列的探查可以递归处理。

【接口】

int quick_select(int a[ ],int n,int k);

复杂度：$O(n)$

输　入：a　整数序列

n　序列长度

k　当前所查询的序数

输　出：所查询的值

【代码】

```
void quickSelect(int a[],int l,int r,int rank)
{
    int i=l,j=r,mid=a[(l+r)/2];
    do{
        while (a[i]<mid) ++i;
        while (a[j]>mid) --j;
        if (i<=j){
            swap(a[i],a[j]);
            ++i;--j;
        }
    }while (i<=j);
    if (l<=j && rank<=j-l+1) quickSelect(a,l,j,rank);
    if (i<=r && rank>=i-l+1) quickSelect(a,i,r,rank-(i-l));
```

```
}

int quick_select(int a[],int n,int k)
{
    quickSelect(a,1,n,k);
    return a[k-1];
}
```

【使用范例】

参见程序 SELECT.CPP。

5.5.2 逆序对

【任务】

给定一个长度为n的数列a，满足a中的元素$a[0]$~$a[n-1]$两两不同，问：存在多少对(i,j)，满足$0 \leqslant i < j \leqslant n-1$且$a[i] > a[j]$。

【说明】

我们用归并排序的方法求数列的逆序对数。

对于数列$a[0\sim n-1]$，首先把$a\left[0\sim\frac{n}{2}\right]$和$a\left[\frac{n}{2}+1\sim n-1\right]$分别排序并求出这两段中的逆序对个数$s_1$和$s_2$。然后再把两段已经排好序的数列合并起来，对于$a\left[0\sim\frac{n}{2}\right]$中的每个数$a[i]$，在合并时统计$a\left[\frac{n}{2}+1\sim n-1\right]$中排在$a[i]$在前面的数的个数，设有$c_i$个，即为$a[i]$和$a\left[\frac{n}{2}+1\sim n-1\right]$组成的逆序对的个数，那么逆序对的总个数为$s_1+s_2+\sum_{i=0}^{\frac{n}{2}}c_i$。

【接口】

long long inversed_pair (int a[], int n);

复杂度：O($n$log$n$)

输　入：a[]　　给定的数列

　　　　n　　　数列的长度

输　出：数列a的逆序对个数

【代码】

```
long long inversed_pair(int a[], int n) {
    if (n == 1) return 0;
    long long sum = 0;
    int mid = n / 2;
    sum += sort(a, mid);
    sum += sort(a + mid, n - mid);
    int *b = new int[n];
    memcpy(b, a, n * sizeof(int));
    for (int i1 = 0, i2 = mid, i = 0; i1 < mid || i2 < n; ++i) {
        if (i2 == n) {
            a[i] = b[i1];
            ++i1;
            sum += i2 - mid;
        } else if (i1 == mid) {
            a[i] = b[i2];
            ++i2;
        } else if (b[i1] < b[i2]) {
            a[i] = b[i1];
            ++i1;
            sum += i2 - mid;
        } else {
            a[i] = b[i2];
            ++i2;
        }
    }
    delete [] b;
    return sum;
}
```

【使用范例】

参见程序 POJ2299.CPP。

5.5.3　最长公共子序列

【任务】

给定两个序列，求他们的最长公共子序列。序列$A_1, A_2, \cdots, A_N$的一个子序列是指

$A_{K_1}, A_{K_2}, \cdots, A_{K_m}$，其中$K_1 < K_2 < \cdots < K_m$。

【说明】

用 3 个数组来完成这个算法。

（1）$A[i]$表示第一个序列。

（2）$B[i]$表示第二个序列。

（3）$dp[i][j]$表示序列A的前i个和序列B的前j个的最长公共子序列。

状态转移方程为

$$dp[i][j] = \max\{dp[i-1][j], dp[i][j-1]\}$$

若$A[i] = B[j]$，$dp[i][j] = \max\{dp[i][j], dp[i-1][j-1] + 1\}$

【接口】

int LCS(int n1, int n2, int A[ ], int B[ ]);

复杂度：$O(n^2)$

输　入：$n1, n2$　　两个序列的长度

　　　　A, B　　两个序列

输　出：最长公共子序列的长度

【代码】

```
int dp[maxn + 1][maxn + 1];

int LCS(int n1, int n2, int A[], int B[])
{
    for (int i = 1; i <= n1; i++)
        for (int j = 1; j <= n2; j++) {
            dp[i][j] = dp[i - 1][j];
            if (dp[i][j - 1] > dp[i][j]) dp[i][j] = dp[i][j - 1];
            if (A[i] == B[j] && dp[i - 1][j - 1] + 1 > dp[i][j])
                dp[i][j] = dp[i - 1][j - 1] + 1;
        }
    return dp[n1][n2];
}
```

【使用范例】

参见程序 POJ1458.CPP。

5.5.4 最长公共上升子序列

【任务】

给你两个序列A,B，求他们的最长公共上升子序列。

【说明】

$dp[i][j]$表示$A[1..i]$和$B[1..j]$的公共上升子序列中以$B[j]$结尾的最长的的长度。

如果$A[i] \neq B[j]$，则$dp[i][j] = dp[i-1][j]$

如果$A[i] = B[j]$，则$dp[i][j] = \max(dp[i][k] + 1)$，其中$k < j$且$A[i] > B[k]$。

朴素实现复杂度为n^3，可以优化到n^2。

【接口】

void getLcis( );

复杂度：$O(n^2)$

输　入：$n1,n2$　　全局变量，两个序列的长度

　　　　A,B　　全局变量，两个序列

输　出：ans　　全局变量，最长公共上升子序列的长度值

　　　　$lcis$　　全局变量，最长公共上升子序列

【代码】

```
int n1, n2;
int A[maxn + 1], B[maxn + 1];
int dp[maxn + 1][maxn + 1];
int pre[maxn + 1][maxn + 1];
int lcis[maxn + 1];

void getLcis()
{
    memset(dp, 0, sizeof(dp));
    memset(pre, 0, sizeof(pre));
    for (int i = 1; i <= n1; i++) {
        int k = 0;
        for (int j = 1; j <= n2; j++) {
            if (A[i] != B[j]) dp[i][j] = dp[i - 1][j];
            if (A[i] > B[j] && dp[i][j] > dp[i][k]) k = j;
            if (A[i] == B[j]) {
                dp[i][j] = dp[i][k] + 1;
```

```
                    pre[i][j] = k;
                }
            }
        }
        int ans = -1, x = n1, y = 0;
        for (int i = 1; i <= n2; i++)
            if (dp[n1][i] > ans) {
                ans = dp[n1][i];
                y = i;
            }
        int cnt = 1;
        while (dp[x][y]) {
            if (A[x] != B[y])
                x--;
            else {
                lcis[ans - cnt] = B[y];
                cnt++;
                y = pre[x][y];
            }
        }
    }
```

【使用范例】

参见程序 POJ2127.CPP。

第二部分

贴　　士

第 6 章　代　　数

6.1　Bertrand 猜想

对任意$n > 3$，都存在$n < p < 2 \times n$，其中p为质数。

6.2　差 分 序 列

序列差分表由它的第0行确定，也就是原序列，但同时也可以由第0条对角线上的元素确定。

换句话说，由差分表的第0条对角线就可以确定原序列。有这样两个公式：

原序列为h_i，第0条对角线为$c_0, c_1, \cdots ,0,0,0, \cdots$

则$h_n = c_0 \times C_n^0 + c_1 \times C_n^1 + \cdots + c_p \times C_n^p$，

$$\sum_0^n h_k = c_0 \times C_{n+1}^1 + c_1 \times C_{n+1}^2 + \cdots + c_p \times C_{n+1}^{p+1}$$

记住这两个公式，差分表（的第0条对角线）就变得非常有用了。

6.3　威尔逊定理

$(p-1)! \equiv -1 \pmod p$，其中p为素数。

6.4　约 数 个 数

要求一个数的约数个数，设它分解质因数的结果为${p_1}^{a_1} \times {p_2}^{a_2} \times \cdots \times {p_m}^{a_m}$（$p_i$为质因数底数，$a_i$为其对应的指数），那么该数的约数个数为$(a_1 + 1) \times (a_2 + 1) \times \cdots \times (a_m + 1)$。

6.5 行列式的值

一般来说，行列式的值的计算，是用初等变换来让原行列式变成上三角（或下三角）矩阵，然后直接由对角线乘积得到。也有时候是通过选取n个行列互不相同的位置，计算它们的乘积之和来得到，在计算和的时候要考虑到正负号，它是通过选取位置在按行排列的情况下列号的逆序对数判定的。

6.6 最小二乘法

对于一个可以化成最小二乘法的问题，标准形式为

$$\min_{x_0,x_1}\left\|\begin{pmatrix}1 & t_1\\ \vdots & \vdots\\ 1 & t_n\end{pmatrix}\begin{pmatrix}x_0\\ x_1\end{pmatrix}-\begin{pmatrix}y_1\\ \vdots\\ y_n\end{pmatrix}\right\|^2=\min_x\|Ax-b\|^2$$

直接得到该式的参数解：

$$x_1=\frac{\sum_{i=1}^{n}t_iy_i-n\cdot\overline{t}\,\overline{y}}{\sum_{i=1}^{n}t_i^2-n\cdot(\overline{t})^2},x_0=\overline{y}-x_1\overline{t}$$

第 7 章 解析几何

7.1 四边形

四边形有如下几个重要性质（D_1, D_2为对角线，M为对角线中点连线，A为对角线夹角）：

1. $a^2 + b^2 + c^2 + d^2 = {D_1}^2 + {D_2}^2 + 4M^2$
2. $S = D_1 D_2 \sin(A)/2$

（以下对圆的内接四边形）

3. $ac + bd = D_1 D_2$
4. $S = \sqrt{((P-a)(P-b)(P-c)(P-d))}$，$P$为半周长

7.2 抛物线

标准方程：$y^2 = 2px$

曲率半径：$R = ((p + 2x)^{\frac{3}{2}})/\sqrt{p}$

弧长：设$M(x, y)$是抛物线上一点，则 $L_{OM} = \frac{p}{2}\left[\sqrt{\frac{2x}{p}\left(1+\frac{2x}{p}\right)} + \ln\left(\sqrt{\frac{2x}{p}} + \sqrt{1+\frac{2x}{p}}\right)\right]$

弓形面积：设M, D是抛物线上两点，且分居一、四象限。作一条平行于MD且与抛物线相切的直线L，若M到L的距离为h，则有$S_{MOD} = \frac{3}{2} MD \cdot h$。

7.3 双曲线

双曲线的定义是到两个定点的距离之差的绝对值为常数的点的轨迹，这两个点称为双曲线的焦点。由这个定义可以导出其他的定义。双曲线关于坐标轴和原点对称。

双曲线的中心在原点时，它的表达式可以写作$x^2/a^2 - y^2/b^2 = 1$。

7.4 椭 圆

椭圆有如下重要性质：

椭圆$\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$，其中离心率$e=\frac{c}{a}$，$c=\sqrt{a^2-b^2}$，焦点参数$p=\frac{b^2}{a}$。

椭圆上(x,y)点处的曲率半径为$R=a^2b^2\left(\frac{x^2}{a^4}+\frac{y^2}{b^4}\right)^{\frac{3}{2}}=\frac{(r_1r_2)^{\frac{3}{2}}}{ab}$，其中$r_1$和$r_2$分别为$(x,y)$与两焦点$F_1$和$F_2$的距离。设点$A$和点$M$的坐标分别为$(a,0)$和$(x,y)$，则$AM$的弧长为$L_{AM}=a\int_0^{\arccos\frac{x}{a}}\sqrt{1-e^2\cos^2 t}\,\mathrm{d}t=a\int_{\arccos\frac{x}{a}}^{\frac{\pi}{2}}\sqrt{1-e^2\sin^2 t}\,\mathrm{d}t$。

椭圆的周长为$L=4a\int_0^{\frac{\pi}{2}}\sqrt{1-e^2\sin^2 t}\,\mathrm{d}t=4aE\left(e,\frac{\pi}{2}\right)$，其中$E\left(e,\frac{\pi}{2}\right)=\frac{\pi}{2}\left[1-\left(\frac{1}{2}\right)^2e^2-\left(\frac{1\times3}{2\times4}\right)^2\frac{e^4}{3}-\left(\frac{1\times3\times5}{2\times4\times6}\right)^2\frac{e^6}{5}-\cdots\right]$

设椭圆上点$M(x,y)$，$N(x,-y)$，$x,y>0$，$A(a,0)$，原点$O(0,0)$，则有：

扇形OAM的面积为$S_{OAM}=\frac{1}{2}ab\arccos\frac{x}{a}$，弓形$MAN$的面积为$S_{MAN}=ab\arccos\frac{x}{a}-xy$。

5个点可以确定一个圆锥曲线。

θ为(x,y)点关于椭圆中心的极角，r为(x,y)到椭圆中心的距离，椭圆极坐标方程：$x=r\cos\theta$，$y=r\sin\theta$，其中$r^2=\frac{b^2a^2}{b^2\cos^2\theta+a^2\sin^2\theta}$。

第 8 章 平面立体几何

8.1 费 马 点

寻找三角形内一个点，使得三个顶点到该点（费马点）的距离之和最小。求法如下：

若有一个角大于等于120°，那么这个点就是费马点。

若不存在，那么对三角形ABC，任取两条边（这里取AB、BC两条），向三角形外做等边三角形，得到的新的顶点称为C'和A'，那么AA'和CC'的交点即是费马点。

8.2 皮 克 定 理

给定坐标均是整点的简单多边形，设其面积为S，内部整点数为a，边界上整点数为b，那么它们满足关系$S = a + b/2 - 1$。

8.3 三 角 公 式

一些常见的公式如下：

$$\sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta$$

$$\cos(\alpha \pm \beta) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta$$

$$\tan(\alpha \pm \beta) = \frac{\tan(\alpha) \pm \tan(\beta)}{1 \mp \tan(\alpha)\tan(\beta)}$$

$$\tan(\alpha) \pm \tan(\beta) = \frac{\sin(\alpha \pm \beta)}{\cos(\alpha)\cos(\beta)}$$

$$\sin(\alpha) + \sin(\beta) = 2\sin\frac{(\alpha + \beta)}{2}\cos\frac{(\alpha - \beta)}{2}$$

$$\sin(\alpha) - \sin(\beta) = 2\cos\frac{(\alpha + \beta)}{2}\sin\frac{(\alpha - \beta)}{2}$$

$$\cos(\alpha) + \cos(\beta) = 2\cos\frac{(\alpha + \beta)}{2}\cos\frac{(\alpha - \beta)}{2}$$

$$\cos(\alpha) - \cos(\beta) = -2\sin\frac{(\alpha+\beta)}{2}\sin\frac{(\alpha-\beta)}{2}$$

$$\sin(n\alpha) = n\cos^{n-1}\alpha\sin\alpha - \binom{n}{3}\cos^{n-3}\alpha\sin^3\alpha + \binom{n}{5}\cos^{n-5}\alpha\sin^5\alpha - \cdots$$

$$\cos(n\alpha) = \cos^n\alpha - \binom{n}{2}\cos^{n-2}\alpha\sin^2\alpha + \binom{n}{4}\cos^{n-4}\alpha\sin^4\alpha - \cdots$$

8.4 三维几何体

棱柱：

1. 体积 $V = Ah$，A为底面积，h为高
2. 侧面积 $S = lp$，l为棱长，p为直截面周长
3. 全面积 $T = S + 2A$

棱锥：

1. 体积 $V = Ah/3$，A为底面积，h为高

（以下对正棱锥）

2. 侧面积 $S = lp/2$，l为斜高，p为底面周长
3. 全面积 $T = S + A$

棱台：

1. 体积 $V = (A_1 + A_2 + \sqrt{A_1A_2})h/3$，$A_1, A_2$为上下底面积，$h$为高

（以下为正棱台）

2. 侧面积 $S = (p_1 + p_2)l/2$，p_1, p_2为上下底面周长，l为斜高
3. 全面积 $T = S + A_1 + A_2$

8.5 托勒密定理

圆的内接四边形中，两对角线所包矩形的面积等于一组对边所包矩形的面积与另一组对边所包矩形的面积之和。

第 9 章 组 合 数 学

9.1 Catalan 数

Catalan 数的递推关系为：$h(n)=h(0)\times h(n-1)+h(1)\times h(n-2)+\cdots+h(n-1)\times h(0)$（其中$n\geqslant 2$）。

由此递推关系可以得到通项公式：

$$h(n)=\frac{C_{2n}^{n}}{n+1}(n=1,2,3,\cdots)$$

一些常见的 Catalan 数：括号序的个数、凸多边形三角剖分的方案数等。

9.2 组 合 公 式

一些常用的组合公式：

$$\sum_{k=1}^{n}(2k-1)^2=\frac{n(4n^2-1)}{3}$$

$$\sum_{k=1}^{n}k^3=\left(\frac{n(n+1)}{2}\right)^2$$

$$\sum_{k=1}^{n}(2k-1)^3=n^2(2n^2-1)$$

$$\sum_{k=1}^{n}k^4=\frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

$$\sum_{k=1}^{n}k^5=\frac{n^2(n+1)^2(2n^2+2n-1)}{12}$$

$$\sum_{k=1}^{n}k(k+1)=\frac{n(n+1)(n+2)}{3}$$

$$\sum_{k=1}^{n} k(k+1)(k+2) = \frac{n(n+1)(n+2)(n+3)}{4}$$

$$\sum_{k=1}^{n} k(k+1)(k+2)(k+3) = \frac{n(n+1)(n+2)(n+3)(n+4)}{5}$$

错排公式：

$$D_n = n!\left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \cdots + \frac{(-1)^n}{n!}\right) = (n-1)(D_{n-2} - D_{n-1})$$

第 10 章　图　　论

10.1　树的计数

有根树的计数：

首先，令 $S_{n,j}=\sum_{1\leqslant i\leqslant n/j}a_{n+1-ij}=s_{n-j,j}+a_{n+1-j}$；

于是，$n+1$个结点的有根树的总数为：$a_{n+1}=\dfrac{\sum_{1\leqslant j\leqslant n}ja_js_{n,j}}{n}$。

注：$a_1=1,a_2=1,a_3=2,a_4=4,a_5=9,a_6=20,a_9=286,a_{11}=1842$

无根树的计数：

当n是奇数时，则有 $a_n-\sum_{1\leqslant i\leqslant n/2}a_ia_{n-i}$ 种不同的无根树；

当n是偶数时，则有 $a_n-\sum_{1\leqslant i\leqslant n/2}a_ia_{n-i}+\frac{1}{2}a_{n/2}(a_{n/2}+1)$ 种不同的无根树。

生成树的计数：

完全图的生成树个数：n^{n-2}；

任意图的生成树个数：生成树计数行列式$tab[i][i]=D_i$，D_i为i的度数；$tab[i][j]=-k$，k为i和j之间的边数。任去一行一列之后的行列式即为答案。

10.2　有特殊条件的汉米尔顿回路

当图有保证任何点的度数都大于点数的一半时，可以构造出原图的汉米尔顿回路。

首先找到一条长度大于点数一半的路径，然后不断做以下两个操作直到得到回路：

（1）化链为环：由于有特殊条件的存在，所以容易证明可以找到一对在链上相邻的点，它们和链的首尾相连。那么就找到了一个环。

（2）破环为链：还是由于条件的存在，此时就可以找到一个还未在环上并且和环上的某点相连的点，把它作为链头就可以得到一条长度加一的链了。

因为长度是单调递增的，所以一定有一个时刻可以得到原图的汉米尔顿回路。

10.3 普吕弗序列

标号树的普吕弗序列是由其对应的树唯一生成的序列。N个顶点的标号树有长度为$N-2$的普吕弗序列。它的求法是每次去掉树上标号最小的叶子节点，然后把普吕弗编码的第i项成为与这个叶子节点唯一相邻的顶点的编号。

由它经常能导出一些树的计数问题，如给定每个顶点的度数，那么此时就相当于给定它们在普吕弗序列中出现的次数，此时要求生成树计数的话就只需要用简单的排列组合方法了。

10.4 模 2 意义下的二分图匹配数

将图邻接矩阵设为g，容易发现每个二分图匹配事实上对应了一个矩阵g中行列各不相同的位置，这恰好让我们联想到矩阵的行列式。又观察到对于矩阵g中任意行列各不相同的位置如果都为1的话，实质上也就是找到了一个二分图匹配，由于在$\bmod 2$意义下-1和1相同，所以得出了结论：$\bmod 2$意义下的二分图匹配数与该图的行列式的值相同。

第 11 章 积　分　表

$(\arcsin x)' = \dfrac{1}{\sqrt{1-x^2}}$	$(\arccos x)' = -\dfrac{1}{\sqrt{1-x^2}}$	$(\arctan x)' = \dfrac{1}{1+x^2}$
$a^x \to a^x/\ln a$	$\sin x \to -\cos x$	$\cos x \to \sin x$
$\tan x \to -\ln\cos x$	$\sec x \to \ln\tan(x/2+\pi/4)$	$\tan^2 x \to \tan x - x$
$\csc x \to \ln\tan\dfrac{x}{2}$	$\sin^2 x \to \dfrac{x}{2}-\dfrac{1}{2}\sin x\cos x$	$\cos^2 x \to \dfrac{x}{2}+\dfrac{1}{2}\sin x\cos x$
$\sec^2 x \to \tan x$	$\dfrac{1}{\sqrt{a^2-x^2}} \to \arcsin\left(\dfrac{x}{a}\right)$	$\csc^2 x \to -\cot x$

$\dfrac{1}{a^2-x^2}(\lvert x\rvert<\lvert a\rvert) \to \dfrac{1}{2a}\ln\dfrac{(a+x)}{a-x}$	$\dfrac{1}{x^2-a^2}(\lvert x\rvert>\lvert a\rvert) \to \dfrac{1}{2a}\ln\dfrac{(x-a)}{x+a}$
$\sqrt{a^2-x^2} \to \dfrac{x}{2}\sqrt{a^2-x^2}+\dfrac{a^2}{2}\arcsin\dfrac{x}{a}$	$\dfrac{1}{\sqrt{x^2+a^2}} \to \ln\left(x+\sqrt{a^2+x^2}\right)$
$\sqrt{a^2+x^2} \to \dfrac{x}{2}\sqrt{a^2+x^2}+\dfrac{a^2}{2}\ln\left(x+\sqrt{a^2+x^2}\right)$	$\dfrac{1}{\sqrt{x^2-a^2}} \to \ln\left(x+\sqrt{x^2-a^2}\right)$
$\sqrt{x^2-a^2} \to \dfrac{x}{2}\sqrt{x^2-a^2}-\dfrac{a^2}{2}\ln\left(x+\sqrt{x^2-a^2}\right)$	$\dfrac{1}{x\sqrt{a^2-x^2}} \to -\dfrac{1}{a}\ln\dfrac{a+\sqrt{a^2-x^2}}{x}$
$\dfrac{1}{x\sqrt{x^2-a^2}} \to \dfrac{1}{a}\arccos\dfrac{a}{x}$	$\dfrac{1}{x\sqrt{a^2+x^2}} \to -\dfrac{1}{a}\ln\dfrac{a+\sqrt{a^2+x^2}}{x}$
$\dfrac{1}{\sqrt{2ax-x^2}} \to \arccos(1-\dfrac{x}{a})$	$\dfrac{x}{ax+b} \to \dfrac{x}{a}-\dfrac{b}{a^2}\ln(ax+b)$

$\sqrt{2ax-x^2} \to \dfrac{x-a}{2}\sqrt{2ax-x^2}+\dfrac{a^2}{2}\arcsin(\dfrac{x}{a}-1)$

$\dfrac{1}{x\sqrt{ax+b}}(b<0) \to \dfrac{2}{\sqrt{-b}}\arctan\sqrt{\dfrac{ax+b}{-b}}$	$x\sqrt{ax+b} \to \dfrac{2(3ax-2b)}{15a^2}(ax+b)^{\frac{3}{2}}$
$\dfrac{1}{x\sqrt{ax+b}}(b>0) \to \dfrac{1}{\sqrt{-b}}\ln\dfrac{\sqrt{ax+b}-\sqrt{b}}{\sqrt{ax+b}+\sqrt{b}}$	$\dfrac{x}{\sqrt{ax+b}} \to \dfrac{2(ax-2b)}{3a^2}\sqrt{ax+b}$
$\dfrac{1}{x^2\sqrt{ax+b}} \to -\dfrac{\sqrt{ax+b}}{bx}-\dfrac{a}{2b}\int\dfrac{\mathrm{d}x}{x\sqrt{ax+b}}$	$\dfrac{\sqrt{ax+b}}{x} \to 2\sqrt{ax+b}+b\int\dfrac{\mathrm{d}x}{x\sqrt{ax+b}}$

续表

$\frac{1}{\sqrt{(ax+b)^n}}(n>2)\to\frac{-2}{a(n-2)}\cdot\frac{1}{\sqrt{(ax+b)^{n-2}}}$		
$\frac{1}{ax^2+c}(a>0,c>0)\to\frac{1}{\sqrt{ac}}\arctan(x\sqrt{\frac{a}{c}})$	$\frac{x}{ax^2+c}\to\frac{1}{2a}\ln(ax^2+c)$	
$\frac{1}{ax^2+c}(a+,c-)\to\frac{1}{2\sqrt{-ac}}\ln\frac{x\sqrt{a}-\sqrt{-c}}{x\sqrt{a}+\sqrt{-c}}$	$\frac{1}{x(ax^2+c)}\to\frac{1}{2c}\ln\frac{x^2}{ax^2+c}$	
$\frac{1}{ax^2+c}(a-,c+)\to\frac{1}{2\sqrt{-ac}}\ln\frac{\sqrt{c}+x\sqrt{-a}}{\sqrt{c}-x\sqrt{-a}}$	$x\sqrt{ax^2+c}\to\frac{1}{3a}\sqrt{(ax^2+c)^3}$	
$\frac{1}{(ax^2+c)^n}(n>1)\to\frac{x}{2c(n-1)(ax^2+c)^{n-1}}+\frac{2n-3}{2c(n-1)}\int\frac{dx}{(ax^2+c)^{n-1}}$		
$\frac{x^n}{ax^2+c}(n\neq1)\to\frac{x^{n-1}}{a(n-1)}-\frac{c}{a}\int\frac{x^{n-2}}{ax^2+c}dx$	$\frac{1}{x^2(ax^2+c)}\to\frac{-1}{cx}-\frac{a}{c}\int\frac{dx}{ax^2+c}$	
$\frac{1}{x^2(ax^2+c)^n}(n\geqslant2)\to\frac{1}{c}\int\frac{dx}{x^2(ax^2+c)^{n-1}}-\frac{a}{c}\int\frac{dx}{(ax^2+c)^n}$		
$\sqrt{ax^2+c}(a>0)\to\frac{x}{2}\sqrt{ax^2+c}+\frac{c}{2\sqrt{a}}\ln\left(x\sqrt{a}+\sqrt{ax^2+c}\right)$		
$\sqrt{ax^2+c}(a<0)\to\frac{x}{2}\sqrt{ax^2+c}+\frac{c}{2\sqrt{-a}}\arcsin\left(x\sqrt{\frac{-a}{c}}\right)$	$\frac{1}{\sqrt{ax^2+c}}(a<0)\to\frac{1}{\sqrt{-a}}\arcsin\left(x\sqrt{-\frac{a}{c}}\right)$	
$\frac{1}{\sqrt{ax^2+c}}(a>0)\to\frac{1}{\sqrt{a}}\ln\left(x\sqrt{a}+\sqrt{ax^2+c}\right)$		
$\sin^2 ax\to\frac{x}{2}-\frac{1}{4a}\sin 2ax$	$\cos^2 ax\to\frac{x}{2}+\frac{1}{4a}\sin 2ax$	$\frac{1}{\sin ax}\to\frac{1}{a}\ln\tan\frac{ax}{2}$
$\frac{1}{\cos^2 ax}\to\frac{1}{a}\tan ax$	$\frac{1}{\cos ax}\to\frac{1}{a}\ln\tan\left(\frac{\pi}{4}+\frac{ax}{2}\right)$	$\ln(ax)\to x\ln(ax)-x$
$\frac{1}{\sin^2 ax}\to-\frac{1}{a}\cot ax$	$x\ln(ax)\to\frac{x^2}{2}\ln(ax)-\frac{x^2}{4}$	$\cos ax\to\frac{1}{a}\sin ax$
$\sin^3 ax\to\frac{-1}{a}\cos ax+\frac{1}{3a}\cos^3 ax$	$\cos^3 ax\to\frac{1}{a}\sin ax-\frac{1}{3a}\sin^3 ax$	
$x^2e^{ax}\to\frac{e^{ax}}{a^3}(a^2x^2-2ax+2)$	$(\ln(ax))^2\to x(\ln(ax))^2-2x\ln(ax)+2x$	
$x^2\ln(ax)\to\frac{x^3}{3}\ln(ax)-\frac{x^3}{9}$	$x^n\ln(ax)\to\frac{x^{n+1}}{n+1}\ln(ax)-\frac{x^{n+1}}{(n+1)^2}$	
$\sin(\ln ax)\to\frac{x}{2}[\sin(\ln ax)-\cos(\ln ax)]$	$\cos(\ln ax)\to\frac{x}{2}[\sin(\ln ax)+\cos(\ln ax)]$	